Automated Testing Tool For UML Behavioral Descriptions

Guntapudi V Ramanaiah
M.Tech., School Of Information Technology
IIT Bombay

March 25, 2000

Acknowledgment

I would like to thank my guides, **Prof. S. Ramesh** and **Prof. Sridhar Iyer** for their invaluable guidance and encouragement which has been always source of inspiration for me.

Guntapudi V Ramanaiah
IIT Bombay
January 8, 2001
Department of Kanwal Rekhi School Of Information Technology
Indian Institute of Technology
Bombay

Abstract

The increasing complexity of safety-critical software systems and the costs associated with software "failure" has given rise to a need for practical and effective testing strategies. Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design, and coding. Testing is a dynamic method for verification and validation, wherein the execution behavior of the given system is observed for several "test cases". The time and cost devoted to testing needs to be managed accurately. Too often, lack of sufficient testing causes schedule and budget over-runs with insufficient guarantee of quality.

An effective software testing strategy starts with the implementation of a component test process. In component testing, critical components (shared, reusable, or complex) are subjected to functional or black box testing to ensure that they are functionally correct. Also, code coverage, i.e., the number of execution paths covered by a given test case, needs to be analyzed to determine the effectiveness of the testing process.

State machines or statecharts are a common approach for the abstraction of the control and data flow in a wide range of application, such as real-time applications, networking protocols, GUI design and safety-critical systems. S-tatechart is a formalism for visual specification of reactive system behavior. The formalism extends traditional finite-state machines with notions of hierarchy and concurrency, and is used in many popular software design notations. A large part of the appeal of using statecharts for testing, is due to the intuitive operational interpretation of state machine behavior. Statecharts are hierarchical in nature and allow one to decompose states into sub-states.

For systems where tasks are performed under time-constraints, the misinterpretation of even a single event in state machine's behavior can lead to collapse the whole underlying system. Hence an efficient testing tool is mandatory for such systems.

In this report, we discuss various testing techniques, testing strategies and statechart semantics, using practical examples. We have developed **StateTest***, a tool that takes a statechart specification and the test scripts as the input, and generates the pass or fail report and the coverage metrics as the output. Presently, StateTest can test the behavior of traditional as well as hierarchical statecharts. The tool can also be extended to test the statecharts that have orthogonal (concurrent execution) as well as history states.*

Contents

Chapter 1

Introduction

1.1 Importance Of Testing

Software testing is a critical element of software quality assurance(SQA) and represents the ultimate review of *specification, design*, and *coding* [28]. Software testing is an expensive and labor-intensive task. It has been estimated that software testing accounts for up to 50 percent of software development, and even in safety-critical systems. If most of software testing could be automated, the cost of software development could be significantly reduced [5].

Since the entire input domain of the program (which in most cases is effectively infinite) cannot be exhaustively searched, formal coverage criteria are sometimes used to determine test inputs. Such coverage criteria help in early fault identification, thereby providing greater assurance of software quality and reliability. Such criteria also provide rules for determining the number of test cases, as well as the repeatability of the test process. Hence, it is necessary to define adequate coverage criteria for testing.

The design of tests for software and other engineered products can be as challenging as the initial design of the product itself. There are three elements in the test design [20].

- **Test Specifications**- Test specifications describe exact inputs to a program together with exact expected outputs. For *example*, LIST has the null string as its single element; expect return value 9.

- **Test Requirements**- Test requirements describe useful sets of input that should be tested. For *example*, LIST must have a single element(value unspecified) and use the null string as some LIST element.

- **Clues**- Clues are the sources for test requirements. For *example*, LIST is a list of strings and a list element is a string.

One of the goals of testing is to execute the program with appropriate inputs, such that design/coding faults become apparent as failures[20]. For instance, consider the following program:

```
void check_square(int do_square, double x)
     {
double y = x ;
if(do_square)
   y = x + x ;
printf("squared value larger? %d\n", y > x);
     }
```

`The fault is that + is used instead of * in statement y = x + x.`

Firstly, to provoke a failure, the program's inputs must cause the faulty statements to be executed. Such inputs need to satisfy a *reachability condition*. In this case, the reachability condition is that **do_square** is true.

Next, the faulty statement must produce a different result than the correct statement. This is called the *necessity condition*. I n this example, $X = 1$ will cause Y to have the incorrect value $2(1 + 1)$ instead of the correct value $1(1 * 1)$. However, $X = 2$ will cause Y to have the correct value 4. The program calculated the right result the wrong way, so it won't fail on that input. The necessity condition for this fault is $X \neq 2 \&\& X \neq 0$.

Finally, the incorrect internal state must propagate so that it becomes visible in the program's results. For example, if X is 4, Y will be 8, instead of 16. However, since both 8 and 16 are greater than 4, the faulty program will print "squared value larger? 1", which is the same as the output of a correct program. In this case, the incorrect internal state was "damped out" before it became visible. Hence, there is a need for a *propagation condition*. The *propagation condition* for the fault requires that $X + X > X$ have a different truth value from $X * X > X$. This is true whenever $X \leq 1$.

All these three conditions are called the **ideal fault conditions**. Hence the program should be tested with inputs satisfying:

```
do_square true AND (X!=0 and X!=2) AND (X<=1).
```

1.2 Statecharts

Statecharts are hierarchical state machines that support the concepts of *orthogonality, aggregation, and generalization*[22]. A behavioral model described as a statechart is based not only on modes and transitions, but also on events, conditions and different types of data items. Statecharts resolve the "blow-up phenomena"[8] associated with state transition diagrams. A realistic system is typically not described as a single statechart but as a hierarchy of statecharts. A hierarchical statechart allows one to decompose states into sub-states. In addition to the hierarchy within a chart, one can use the mechanism of activity-charts to describe the data flow in the system as well its functional decomposition. In this case, the whole system is decomposed to a hierarchy of activities, the data flow between the activities is specified, and the behavior of each of these activities is described using statecharts. This in effect provides a mechanism of decomposing a system into a tree of statecharts, only some of them are active at a given time. The visibility rules associated with statechart models are similar to those found in other top-down structured methods. Elements defined in an activity, can be referred to by any of its sub-activities and statecharts.

Statecharts, or behavioral models in general, can be used to describe more than just a model of the system being designed. One can design a 'watchdog' statechart to observe the system during its operation and monitor its behavior. As a watchdog, it can be executed in parallel to the system model. A statechart watchdog can operate on the inputs and outputs of the whole system. However, it is often very useful to monitor internal values or even inject faults by modifying them.

This report deals with the scoping of elements in a hierarchical statechart model, and its impact on testing.

1.3 Testing Of Statecharts

The overall behavior of the underlying system that is modeled by *statecharts* depends upon the transitions made by the event occurrence. Since even a single event misinterpretation in statechart's behavior can lead to collapse the whole underlying system, an efficient testing tool is mandatory. This is especially true for systems where tasks are performed within time-constraints. Hence appropriate care needs to be taken while modeling the behavior of the system for testing.

Chapter 6 of this report, provides a detailed discussion of the various issues in testing of statecharts.

1.4 Scope Of the Project

Firstly, this report provides a comprehensive view of the details in testing statecharts. We have also developed **StateTest**, a tool for testing of statecharts. StateTest has mainly four modules *StatesInfo, CopyStateHierarchy, TestScriptReading*, and *ReportGenerator*. It takes two inputs from the user. The first input is a file containing the statechart specification and the second input is a file containing the test script (a set of test cases). StateTest applies the given test cases to the statechart specification and generates three outputs. The first output is a *pass or fail* report for each test case, and the second output is the *coverage metrics* (transition, state, event and state-event coverage), while the third is a detailed *error report*. We have tested the StateTest tool for various known input specifications and believe that this StateTest tool can be used for all applications that can be modeled by **statecharts**.

1.5 Organization Of the Report

In the First chapter, Introduction, brief introduction of testing criteria and s-tatecharts is discussed. In the Second chapter, various testing techniques such as *white box* testing and *black box* testing are explored. In the Third chapter, many testing strategies including *unit testing, integration testing, system testing, and validation testing* are discussed. The Fourth chapter shows the *literature survey.* The Fifth, Sixth, and Seventh chapters are totally dedicated to explore the semantics of statecharts, UML(Unified Modeling Language), testing strategy for the statecharts, and *coverage metrics* including transition coverage, event coverage, state coverage and state-event coverage. The Eighth chapter, shows a skeleton algorithm for the testing phase and Object Design Specification for the project. The Nineth chapter shows the **demo results**. The Tenth chapter concludes the report.

Chapter 2

Testing Techniques

2.1 Testing Fundamentals

2.1.1 Testing Objectives

The objective is to design tests that systematically uncover different classes of errors and do so with a minimum amount of time and effort.

- Testing is a process of executing a program with the intent of finding an error.
- A good test is one that has a high probability of finding an as yet undiscovered error.
- A successful test is one that uncovers an as yet undiscovered error.

  Testing cannot show the absence of defects, it can only show that software defects are present[28].

2.1.2 Test Information Flow

Figure 2.1 shows gives a top-level view of the various phases in testing. It includes four phases i.e, *Testing, Evaluation, Reliability model* and *Debug.* Software configuration includes a software requirements specification, a design specification, and source code. A test configuration includes a test plan and procedures, test cases, and testing tools. Software configuration and the test configuration are input to the Testing phase. Test results and expected results are input to the Evaluation phase. In this phase, test results and expected results are compared and generated the error report. In the Debug phase, finding are corrected. Reliability phase is used to determine the reliability of the underlying system. It is difficult to predict the time to debug the code, hence it is difficult to schedule.

2.1.3 Test Case Design

Designing a test case can be as difficult as the initial design. **White box testing** is used to test if a component conforms to its design, while **Black box**

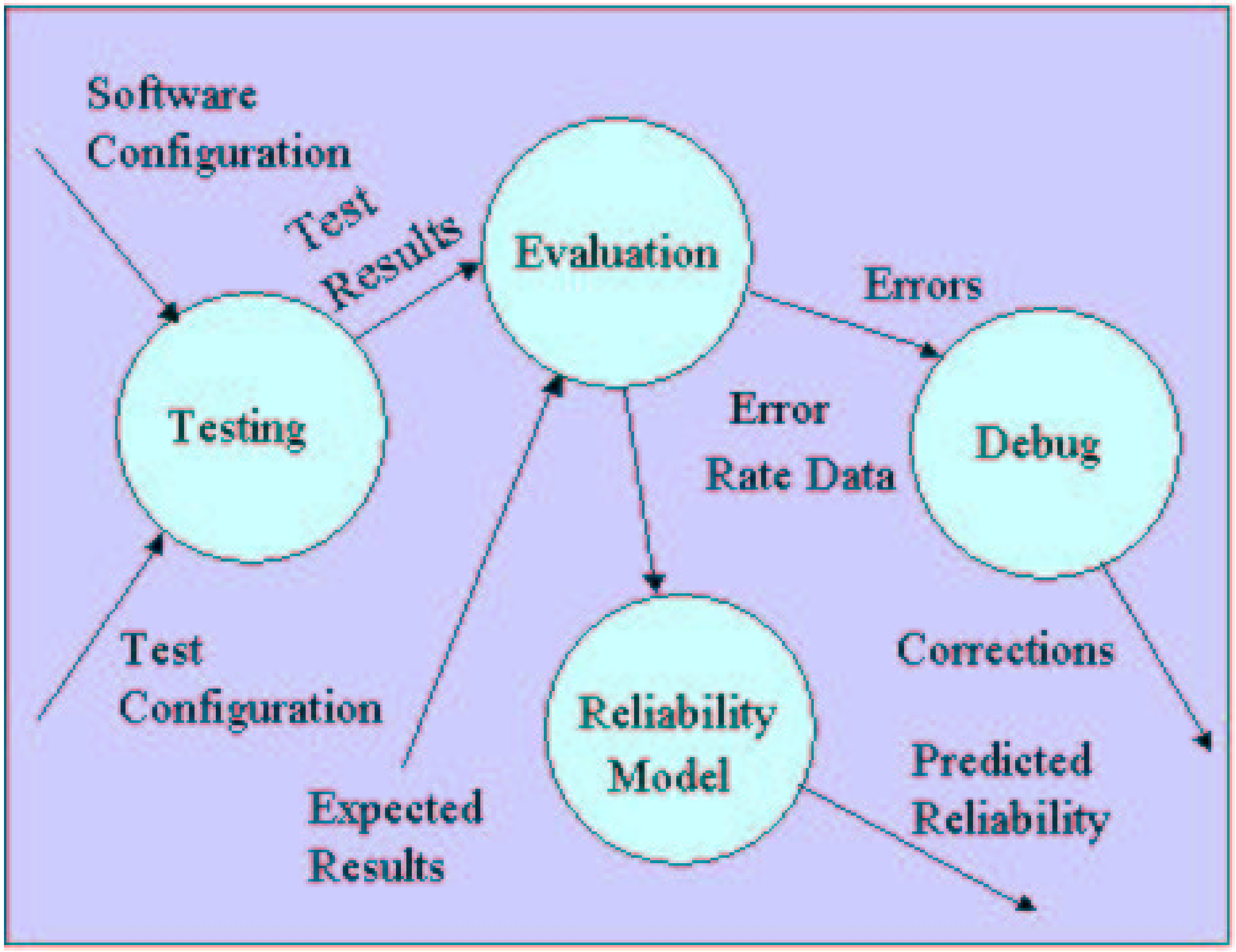

Figure 2.1: DFD for testing

testing is used to test if a component conforms to its specification. Testing approaches in general, cannot prove correctness, as they may not cover all possible execution paths.

2.2 White Box Testing

White-box testing is a test case design method that uses the control structures of a procedural design.

```
Using white-box testing methods one can derive test cases to ensure :
  1 all independent paths are exercised at least once.
  2 all logical decisions are exercised for both true and false paths.
  3 all loops are executed at their boundaries and within operational
    bounds.
  4 all internal data structures are exercised to ensure validity.
```

2.2.1 Basis Path Testing

Basis path testing is a white-box testing technique proposed by Tom McCabe. The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a *basis set* of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing. Any procedural design can be translated into a flow graph. Note that compound boolean expressions at tests generate at least two predicate node and additional arcs.

Cyclomatic Complexity

The cyclomatic complexity gives a quantitative measure of the logical complexity. This value gives the number of independent paths in the basis set, and an upper bound for the number of tests to ensure that each statement is executed at least once.

An independent path is any path through a program that introduces at least one new set of processing statements or a new condition (i.e., a new edge). For example, consider the following program:

```
0.{
1. i = 1;
2. while (i <= n) {
3.   j = i;
4.   while (j <= i) {
5.     if (A[i] < A[j]);
6. swap(A[i],A[j]);
7.     j = j + 1; }
8. i = i + 1;}
9.}
```

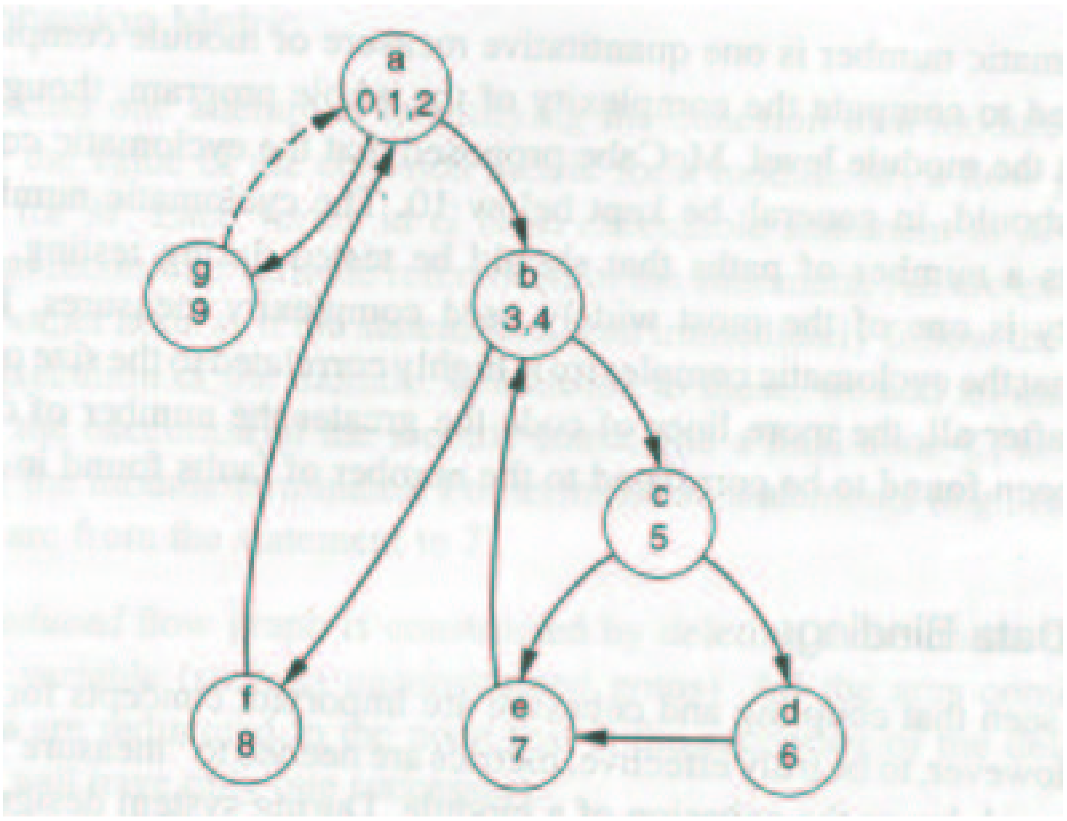

Figure 2.2: Diagram for flow graph of a bubble sort procedure

Cyclomatic Complexity can be calculated by using anyone of the following methods

```
1 Number of regions of flow graph.
2 Number of edges - Number of nodes + 2
3 Number of predicate nodes + 1
```

- Independent Paths are :

```
1 b c e b
2 b c d e b
3 a b f a
4 a g a
```

The corresponding *flow graph* is given by :

Cyclomatic complexity provides upper bound for number of tests required to guarantee coverage of all program statements. The cyclomatic complexity for 'bubble sort' procedure in the figure is $V(G) = 9 - 7 + 2 = 4$.

Deriving Test Cases

```
1 Using the design or code, draw the corresponding flow graph.
```

```
2 Determine the cyclomatic complexity of the flow graph.
3 Determine a basis set of independent paths.
4 Prepare test cases that will force execution of each path in
  the basis set.
```

Graph Matrices

These can automate derivation of flow graph and determination of a set of basis paths. To develop a software tool to do basis path testing, graph matrix can be used [28].

2.2.2 Control Structure Testing

Basic path testing is one example of control structure testing.

Condition Testing

Condition testing aims to exercise all logical conditions in a program module. Condition testing methods focus on testing each condition in the program.

It can define :

- *Relational expression* : (E1 op E2), where E1 and E2 are arithmetic expressions.
- *Simple condition* : Boolean variable or relational expression, possibly preceded by a NOT operator.
- *Compound condition*: composed of two or more simple conditions, boolean operators and parentheses.
- *Boolean expression* : Condition without relational expressions.

Example 1 : C1 = B1 & B2

- where B1, B2 are boolean conditions..
- Condition constraint of form (D1, D2) where D1 and D2 can be true (t) or false(f).
- The branch and relational operator test requires the constraint set (t,t), (f,t), (t,f) to be covered by the execution of C1.

Coverage of the constraint set guarantees the detection of relational operator errors.

Data Flow Testing

In this testing, selects test paths according to the location of definitions and use of variables.

Loop Testing

Loops fundamental to many algorithms. you can define loops as simple, concatenated, nested, and unstructured.

2.3 Black Box Testing

Black-box testing focuses on the functional requirements of the software.It is not an *alternative* to white-box techniques[28, 15]. Rather,it is a complementary approach that is likely to uncover a different classes of errors than white-box methods. It attempts to find :

- incorrect or missing functions
- interface errors
- errors in data structures or external database access
- performance errors
- initialization and termination errors.

2.3.1 Equivalence Partitioning

Equivalence partitioning is a black-box testing method that divides the input domain into classes of data for which test cases can be generated. It attempts to uncover classes of errors. Test case design for equivalence partitioning is based on an evaluation of equivalence classes for an *input condition*. An equivalence class represents a set of valid or invalid states. An input condition is either a specific numeric value, range of values, a set of related values, or a boolean condition. Equivalence classes can be defined by :

- If an input condition specifies a range or a specific value, one valid and two invalid equivalence classes defined.
- If an input condition specifies a boolean or a member of a set, one valid and one invalid equivalence classes defined.

Test cases for each input domain data item developed and executed.

2.3.2 Boundary Value Analysis(BVA)

Large number of errors tend to occur at boundaries of the input domain. BVA leads to selection of test cases that exercise boundary values. BVA complements equivalence partitioning. Rather than select any element in an equivalence class, select those at the 'edge' of the class.

```
Examples :
  1 For a range of values bounded by a and b,
      test (a-1), a, (a+1), (b-1), b, (b+1).
  2 If input conditions specify a number of values n,
     test with (n-1), n and (n+1) input values.
 3 Apply 1 and 2 to output conditions
     (e.g., generate table of minimum and maximum size).
 4 If internal program data structures have boundaries
     (e.g., buffer size, table limits), use input data to
     exercise structures on boundaries.
```

2.3.3 Comparison Testing

In some applications(e.g., aircraft avionics, nuclear power plant control), the reliability of software is absolutely critical. Redundant hardware and software are often used to minimize the possibility of error. when redundant software is developed, separate engineering teams develop independent versions of the software using the same specification[25, 15, 28].

After that test each version with same test data to ensure all provide identical output and run all versions in parallel with a real-time comparison of results. Even when only a single version will be run in final system, these independent versions form the basis of a black-box testing technique called*comparison testing* or *back-to-back testing*. When outputs of versions differ, each is investigated to determine if there is a defect. This method does not catch errors in the specification.

2.4 Summary

In this chapter we have presented the details of two main approaches to testing, viz., white-box testing and black-box testing. Test cases are decided solely on the basis of the requirements or specification of the program or module, and the internals of the module or the program are not considered for selection of test cases. Due to this nature, functional testing often called *black-box testing*. In the structural approach, test cases are generated based on the actual code of the program or module to be tested. This structural approach is called *white-box testing*, also called *glass box testing*. Unlike the criteria for black-box testing, which are frequently imprecise, the criteria for white-box testing are generally quite precise as they are based on program structures, which are formal and precise.

In the next chapter, we discuss several testing strategies.

Chapter 3

Testing Strategies

3.1 A Software Testing Strategy

The generic aspects of a test strategy are :

- Testing begins at the module level and works 'outward'.
- Different testing techniques are used at different points in time.
- Testing conducted by developer and (for larger projects) by an independent test group.
- Testing and Debugging are two different activities, but debugging should be incorporated into any testing strategy.

```
System development proceeds with steps :
  (1) System engineering
  (2) Requirements
  (3) Design
  (4) Coding
```

The software engineering process may be viewed as a *spiral* as shown in the figure[28].

Faults can occur during any phase in the **software development cycle**. Verification is performed on the output of each phase, but some faults are likely to remain undetected by these methods. These faults will be eventually reflected in the code. Testing is usually relied on to detect these faults, in addition to the faults introduced during the code phase itself. Due to this, different levels of testing are used in the testing process. Each level of testing aims to test different aspects of the system. The basis levels are *unit testing, integration testing, validation testing*, and *system testing*. These different levels of testing attempt to detect different types of faults. Unit testing is essentially for verification of the code produced during the coding phase. Integration testing activity can be considered testing the design. Validation testing goal is to see if the software meets its requirements. System testing test the entire software system. Testing here focuses on the external behavior of the system.

The subsequent sections discuss each of these testing strategies in detail.

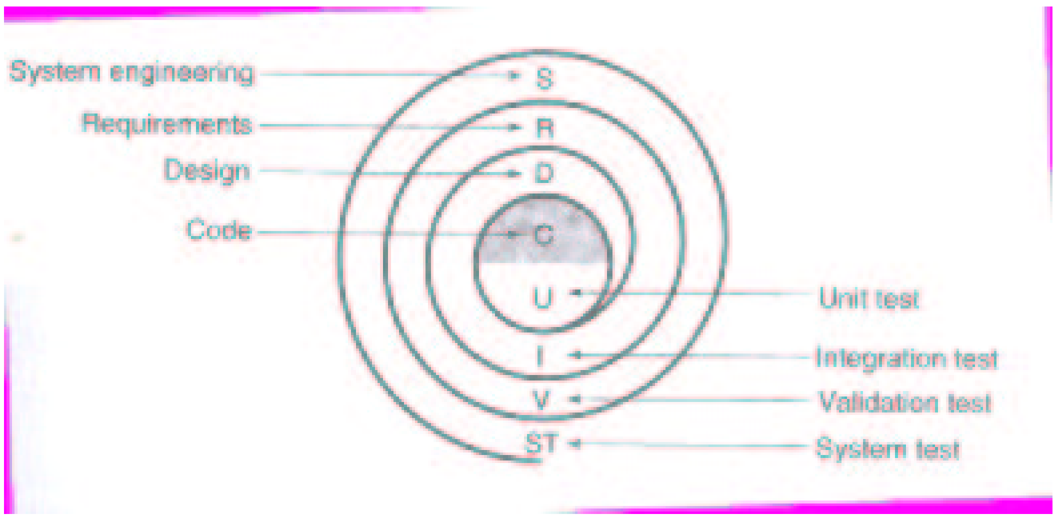

Figure 3.1: Diagram for testing strategy

3.2 Unit Testing

Unit testing does

- Module level testing with heavy use of white box testing techniques.
- Exercise specific paths in the modules control structures for complete coverage and maximum error detection.

Unit testing can test: interface, local data structures, boundary conditions, independent paths, error handling paths.

3.3 Integration Testing

Integration testing is a systematic technique for constructing the program structure while conducting tests to uncover errors associated with interfacing[28, 15]. It addresses the issues associated with the dual problems of verification and program construction. Black-box test case design techniques are the most prevalent during integration, although a limited amount of white-box testing may be used to ensure coverage of major control paths. There are two approaches to integration testing : *non-incremental integration* and *incremental integration*

There is often a tendency to attempt *non-incremental integration*. That means, constructing a program using *big bang* approach. All modules are combined in advance. The entire program is tested as a whole. Usually, a set of errors are encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small segments, where errors are easier to isolate and correct. Interfaces are more likely to be tested completely, and a systematic test approach may be applied.

In this report, we focus on incremental testing. Incremental testing may be done using the approaches described below.

3.3.1 Top Down Integration

Top down integration is an incremental integration testing approach. In this, modules integrated by moving down the program design hierarchy. Either *depth first* or*breadth first* top down integration may be used. It verifies major control and decision points early in design process. Depth first implementation allows a complete function to be implemented, tested and demonstrated. Top down integration forced (to some extent) by some development tools in programs with graphical user interfaces.

3.3.2 Bottom-Up Integration

Bottom up integration is also type of incremental integration approach.

Bottom-up integration testing begins construction and testing with *atomic modules* (lowest level modules). One can use driver program to test.

```
It has the following steps :
  1 Low level modules combined in clusters (builds) that
    perform specific software sub functions.
  2 Driver program developed to test.
  3 Cluster is tested.
  4 Driver programs removed and clusters combined, moving
    upwards in program structure.
```

3.3.3 Comments on Integration Testing

The major disadvantage of: top-down approach is the need for stubs generation and of bottom-up integration is that "the program as an entity does not exist until the last module is added". Critical modules should be tested and integrated early. In general, a combined approach is preferred.

3.4 Validation Testing

Validation testing aims to demonstrate that the software functions in a manner that can be reasonably expected by the customer. It tests conformance of the software to the *Software Requirements Specification.* This should contain a section "Validation Criteria" which is used to develop the validation tests.

3.4.1 Validation Test Criteria

Software validation is achieved through a series of black box tests to demonstrate conformance with requirements. It is used to check that: all functional requirements satisfied, all performance requirements achieved, documentation is correct and 'human-engineered', and other requirements are met (e.g., compatibility, error recovery, maintainability)[28, 15]. When validation tests fail it may be too late to correct the error prior to scheduled delivery. It is often necessary to negotiate a method of resolving deficiencies with the customer.

3.4.2 Configuration Review

An audit to ensure that all elements of the software configuration are properly developed, cataloged, and have the necessary detail to support maintenance.

3.4.3 Alpha and Beta Testing

It is very difficult to anticipate how users will really use software. If there is one customer, a series of *acceptance tests* are conducted (by the customer) to enable the customer to validate all requirements. If software is being developed for use by many customers, can not use acceptance testing. An alternative is to use *alpha* and *beta testing* to uncover errors. *Alpha testing* is conducted at the developer's site by a customer. The customer uses the software with the developer and recording errors and usage problems. Alpha testing conducted in a controlled environment. *Beta testing* is conducted at one or more customer sites by end users. It is 'live' testing in an environment not controlled by the developer. The customer records and reports difficulties and errors at regular intervals[28].

3.5 System Testing

Software is only one component of a system. Software will be incorporated with other system components and system integration and validation tests performed. For software based systems can carry out: *recovery testing,security testing, stress testing, performance testing*

3.6 Debugging

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. Debugging is not testing.

3.7 Summary

In this chapter, we have discussed various testing strategies including unit testing, integration testing, validation testing, and system testing. In the next chapter, we will see the literature survey on software testing tools.

Chapter 4

Literature Survey

This chapter presents the details of four testing tools studied as part of this project. Evaluation/free versions for the first three of these tools were unavailable, hence they were studied from product documentation. Evaluation version was available for the last tool "$+1Software$", which was downloaded and extensively used.

4.1 AdaTest and Cantata

A comprehensive testing strategy for state machine software should set targets for both structural coverage measures as provided by AdaTEST and Cantata and functional measures. The algorithm used to implement structural coverage metrics such as STATEMENT_COVERAGE and DECISION_COVERAGE within AdaTEST and Cantata can be adopted for other metrics[19].

Users of AdaTEST and Cantata can add code to a test script to implement State-Event coverage based on the IPL algorithm:

- Declare a two dimensional array of scores, which one axis being indexed by the state, and the other axis being indexed by the event. Initialize all cells to zero

  Within each test case, write statements to increment the corresponding cell of the array according to the current state and event(not the next state

- At the end of test script, calculate the portion of non-zero cells in the array. Use the CHECK_ANALYSIS command to check the calculated coverage against the objective level

AdaTEST and Cantata assertions can be used to simplify the testing of state machines. To verify State-Event Coverage, assertions can be used to ensure that all events have been exercised in all states.

4.2 Property-Based Testing : Tester's Assistant

Analysts test computer programs to determine if they meet reliability and assurance goals[12]. In other words, testing validates semantic properties of a program's behavior. In order to do this, the actual program must be tested at the source code level, not some higher-level description of the program. However, to validate high-level properties, the properties must be formalized, and the results of the testing related formally to the properties. Property-based testing is a testing methodology that addresses this need. The specification of one or more properties drives the testing process,which assures that the given program meets the stated property. For example, if an analyst wants to validate that a specific program correctly authenticates a user, a property-based testing procedure tests the implementation of the authentication mechanisms in the source code to determine if the code meets the specification of "correctly authenticating user". The steps involved in the property-based testing are :

- first, analyst specifies the target property in a low-level specification language called **TASPEC**(Tester's Assistant SPECification language).
- the program is sliced and code irrelevant to the property disregarded.
- the Tester's Assistant automatically translates the TASPEC specification into a oracle that will check the correctness of program executions with respect to the desired property.
- a new path-based code coverage metric called *"iterative contexts"* efficiently captures the slice-based computations in the program.

4.3 ATTOL Software

The ATTOL company provides the following products :

ATTOL UniTest, ATTOL SystemTest, ATTOL Coverage. *ATTOL UniTest* is the first commercial tool to automate software component testing for C, C++, Ada 83 and 95. It is aimed at professionals wishing to improve the reliability of their while optimizing development costs. *ATTOL SystemTest* is a product aimed at distributed systems developers and integrators who need to test interaction or communication between several sub systems. With it's ability to work with all communication interfaces written in C and C++, ATTOL SystemTest is ideal for testing applications based on commercial (MQSeries, TIB, Tuxedo, Encina, etc.) and proprietary middle ware. *ATTOL Coverage* is easy to use and aimed at developers. It measures test effectiveness by analyzing code coverage.

4.4 +1 Software Engineering

The +1 Software Engineerings products support configuration management, build management, problem reporting, software reuse, software testing, HTML reports, data repository, reverse engineering, and software metrics. It supports the following products:

+1Base supports multiple software projects. After selecting which project to work on, +1Base graphically displays the calling structure of the program including any recursive routines. You can change the model, traverse the calling structure, and edit or view a module's source code, documentation, and test files.

+1CM is an advanced configuration management system supporting identification, variations, baselines, accounting, auditing, and access control. +1CM enhances the Source Code Control System (SCCS), an automated configuration management tool.

+1CR supports problem report management which allows you to submit, list, view, query, print, and administer problem reports. When combined with +1CM, +1CR can support process management. This ensures that when a file is checked in using +1CM, the problem report number exists and its current status allows for check ins.

+1DataTree predefines the following modules: *data elements, data structures, files, external entities, glossary and personnel.*

+1Test supports unit, integration, and regression testing. A unit test tests an individual source code module. Integration testing tests a "build" (i.e., a submodel) of the project. And regression testing runs all currently defined test cases.

+1Reports generates a number of customized, always up-to-date project reports. You can create new reports and load in existing reports. It has options to incorporate, generates your reports in either text or HTML format.

+1Reuse supports reuse repositories created and maintained by the user, project wide "filtered" repositories which can be under strict quality controls, and selective reuse. +1Reuse supports reuse of:*design, documentation, source code, header files, test cases, test shell scripts, expected test results and modeling information.*

This tool was downloaded, tested against several C programs and its output was studied. This gave us a clear understanding of the requirements to be met by a testing tool, thereby enabling us to appropriately design StateTest, which is described in chapter 8.

In the next chapter, we discuss UML, which is a pre-requisite for the understanding of statecharts.

Chapter 5

A Brief Look At UML

5.1 What Is the Unified Modeling Language?

The unified Modeling Language is a language that unifies the industry's best engineering practices for modeling systems. It was originally conceived by Rational Software Corporation and the Three Amigos, Grady Booch (Booch's method expressed in his book *Object-Oriented Analysis and Design*), James Rumbaugh(OMT) and Ivar Jacobson (OOSE : *A Use Case Driven approach*). It is supported by the UML Partners Consortium (Rational Software Corporation, Microsoft Corporation, Hewlett-Packard Company, Oracle Corporation, Sterling Software, MCI Systemhouse Corporation, and ICON Computing). The UML

- Is a *language*. It is not simply notation for drawing diagrams, but a complete language for capturing knowledge (semantics) about a subject and expressing knowledge (syntax) regarding the subject for the purpose of communication

- Applies to *modeling* the systems. Modeling involves a focus on understanding (knowing) a subject (system) and capturing and being able to communicate this knowledge.

- Is the result of *unifying* the information systems and technology industry's best engineering practices (principles, techniques, methods ,and tools) .

- Is used for specifying, visualizing, constructing, and documenting systems.

- Is based on the object-oriented paradigm.

- Is a modeling language for specifying, visualizing, constructing, and documenting the artifacts of a system-intensive process.

- Is an evolutionary general-purpose, broadly applicable, tool-supported, industry-standardized modeling language.

- Applies to a multitude of different types of systems, domains, and methods or processes.

- Enables the capturing, communicating, and leveraging of strategic, tactical, and operational knowledge to facilitate increasing value by increasing quality, reducing costs, and reducing time-to-market while managing risks and being pro-active in regard to ever-increasing change and complexity.

The goals of the UML are to:

- ◦ Be a ready-to-use expressive visual modeling language that is simple and extensible.
- ◦ Have extensibility and specialization mechanisms for extending, rather than modifying, core concepts.
- ◦ Be implementation independent(programming language).
- ◦ Be process independent(development).
- ◦ Encourage the growth of the object-oriented tools market.
- ◦ support higher-level concepts (collaborations, frameworks, patterns, and components).
- ◦ Addressing recurring architectural complexity problems (physical distribution and distributed systems, concurrency and concurrent systems, replication, security, load balancing, and fault tolerance) using component technology, visual programming, patterns, and frameworks.
- ◦ Be scalable.
- ◦ Be widely applicable and usable.
- ◦ Integrate best engineering practices.

5.2 The Building Blocks Of UML

UML defines nine types of diagrams to represent the various modeling viewpoints [6, 22, 11, 7]. A diagram provides the user with the means of visualizing and manipulating model elements. The diagrams may show all or part of the characteristics of the model elements, with a level of detail that is suitable in the context of a given diagram. Diagrams may also gather together pieces of linked information to show, for example, the characteristics inherited by a class.

- **Activity diagrams** represent the behavior of an operation as a set of actions
- **Class diagrams** represent the static structure in terms of classes and relationships
- **Collaboration diagrams** are a spatial representation of objects, links, and interactions

- **Component diagrams** represent the physical components of an application
- **Deployment diagrams** represent the deployment of components on particular pieces of hardware
- **Object diagrams** represent objects and their relationships, and correspond to simplified collaboration diagrams that do not represent message broadcasts
- **Sequence diagrams** are a temporal representation of objects and their interactions
- **Statechart diagrams** represent the behavior of a class in terms of states
- **Use case diagrams** represent the functions of a system from user's point of view

Sequence and collaboration diagrams can be grouped together into **interaction diagrams**.

One of the crucial aspects of the Unified Modeling Language (UML) that makes it so valuable for real-time and embedded systems [10]is its heavy reliance on and support for finite state machines. State machines are critical in the construction of executable models that can effectively react to incoming events in a timely fashion. The UML state machine model represents the current state of the art in state machine theory and notation, all based on David Harel's statecharts. Statecharts describe both how objects communicate and collaborate and how they carry out their own internal behavior. They must also reflect important OO issue like inheritance. State machines are the primary means within the UML for capturing complex dynamic behavior. The UML is a third-generation state-of-the-art object modeling language that defines a comprehensive set of notations, and, more importantly, defines the semantics (meaning) of those language elements. The UML is an inherently discrete language's meaning that it emphasizes discrete representations of dynamic behavior, such as state machines, over continuous representations. Although many object systems do, in fact, perform continuous control functions and do it well, the UML provides special support in the area of finite state machines.

In the next chapter we discuss statecharts in detail before going to their testing issues.

Chapter 6

Statecharts : A Visual Formalism for Complex Systems

6.1 Statechart Diagrams

Statechart diagrams represent state machines from the perspective of states and transitions. For many applications, state machines grow large and cumbersome. Statecharts extend state machines to deal with those problems [20, 22, 19]. The purpose of using statecharts is to formally specify the behavior of the instances of a given class in response to external stimuli[24]. In UML notation, a statechart is represented by the directed graph of states connected by transitions [1, 16]. The origins of statecharts are :

- finite state machines (FSM)
- state transition diagrams
- Harel's statecharts

6.2 Semantics of Statechart

State

- a finite period in the life of an object, when the object satisfies some condition, or performs some action, or waits for some event to occur.

Event

An event is an occurrence of

- a change in truth value of a condition
- a receipt of a message
- the end of a designated period of time

Events cause changes in state (transitions).

Transitions

A state transition can have the following elements associated with it:

- an action (behavior that occurs when the transition takes place)
- a guard (Boolean expression that must be true for the transition to be allowed)

Actions and guards are behaviors, and typically become private operations. State transitions can also trigger events.

Event[Guard]/ Action Transition

- An event prompts the transition between states
- A guard is used to specify that this transition can occur only if the guard is true
- An action is performed when the transition occurs

Internal Activities

Internal activities are performed in response to an event received (on entry, exit, or some other event). Usually, internal activities that result in invocation of*private* operations.

State Details

Actions that accompany state transitions into (out of) a state can be noted as entry (exit) actions within the state. Behavior within a state is called an activity. Activities can be a simple action or an event sent to another object. Activities are optional.

State Detail Notation

A state detail notation is given by :

```
entry: simple action
entry: ^destination class name.event name
do: simple action
do: ^destination class name.event name
exit: simple action
exit: ^destination class name.event name
```

Composite States

A state can be decomposed into :

- a set of mutually exclusive sub-states (or-decomposition)
- a set of concurrent sub-states (and-decomposition)

Special States

- **History State**- a state resumed upon re-entry in the state
- **Activity**- an operation within a state represented by a nested statechart

Figure 6.1 shows the simple statechart and statechart having history state.

6.3 History

By default, a state machine does not have any memory. The special notation **H** offers a mechanism to memorize the sub-state last visited, and to get back to it during a transition entering the encompassing super-state. The history indicator applies to the level in which the **H** symbol is declared. It is also possible to memorizing the last active sub-state, regardless of it's depth; this is indicated by the **H*** symbol. The use of **history** observed in the figure 6.1 to implement a *dishwasher*.

6.4 Guards

A **guard** is a Boolean condition that may or may not validate the triggering of an event occurrence. Guards make it possible to maintain the determinism of a state machine, even when many transitions can be triggered by the same event. When the event takes place, guards, which must be mutually exclusive, are evaluated, and then a transition is validated and triggered. As shown in the figure 6.2, when the stack is not empty, **pop operation** is allowed on the current stack.

6.5 Generalization of States

Statechart diagrams may become rather difficult to read when the number of connections between states become high. The solution for this situation is to apply the principle of state generalization - the more general states are called*superstates*, and the more specific states are called*sub-states*. It is preferable to limit the links between the hierarchical levels of a state machine, by systematically defining an initial (pseudo) state for each level as shown in last diagram in the figure 6.3. In the figure, state **root** is super state for the states **error, state1, and state2**. State1 and state2 are in turn have sub-states (s1, s2) and (p1, p2) respectively.

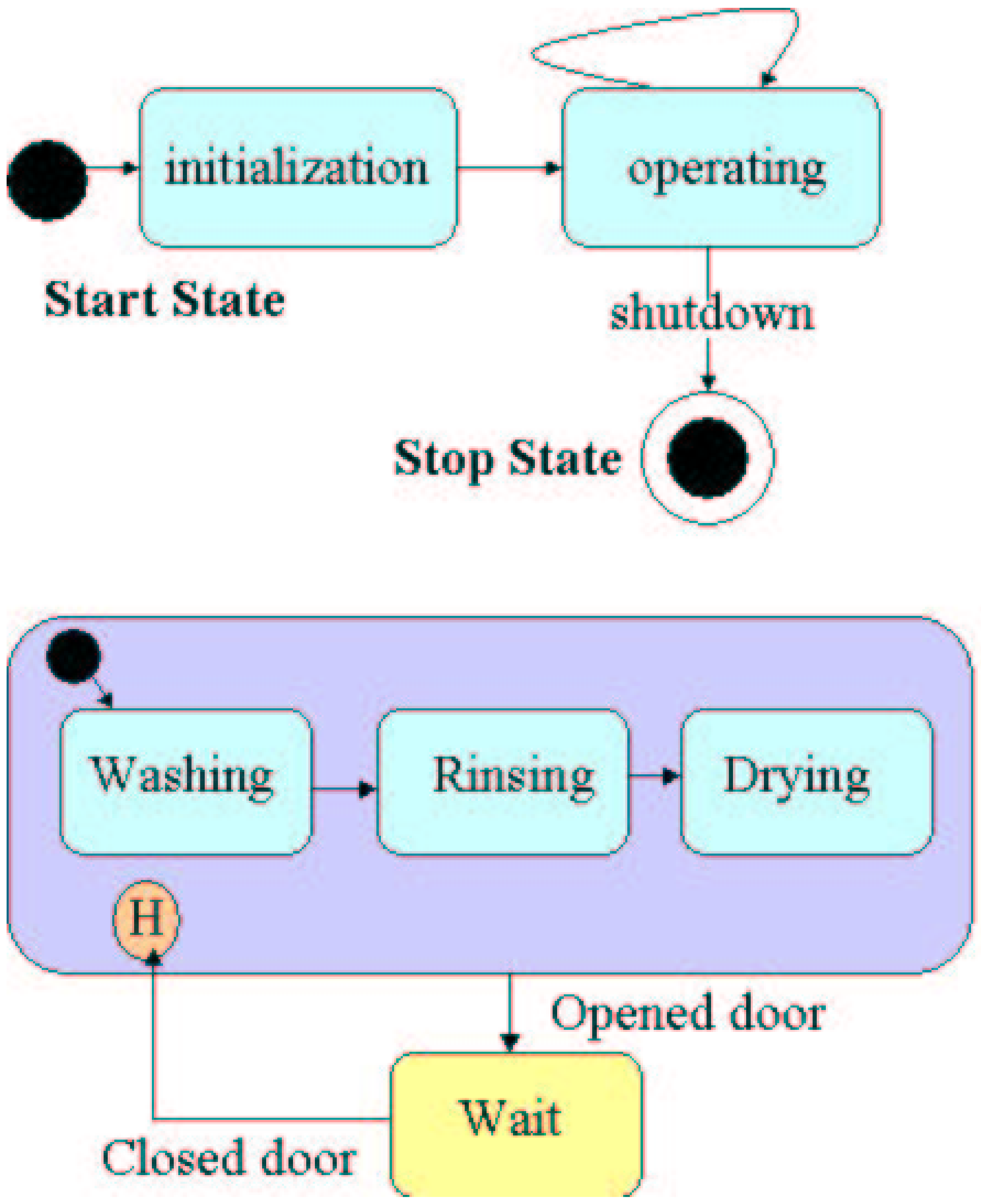

Figure 6.1: Representation of simple statechart and statechart with history state

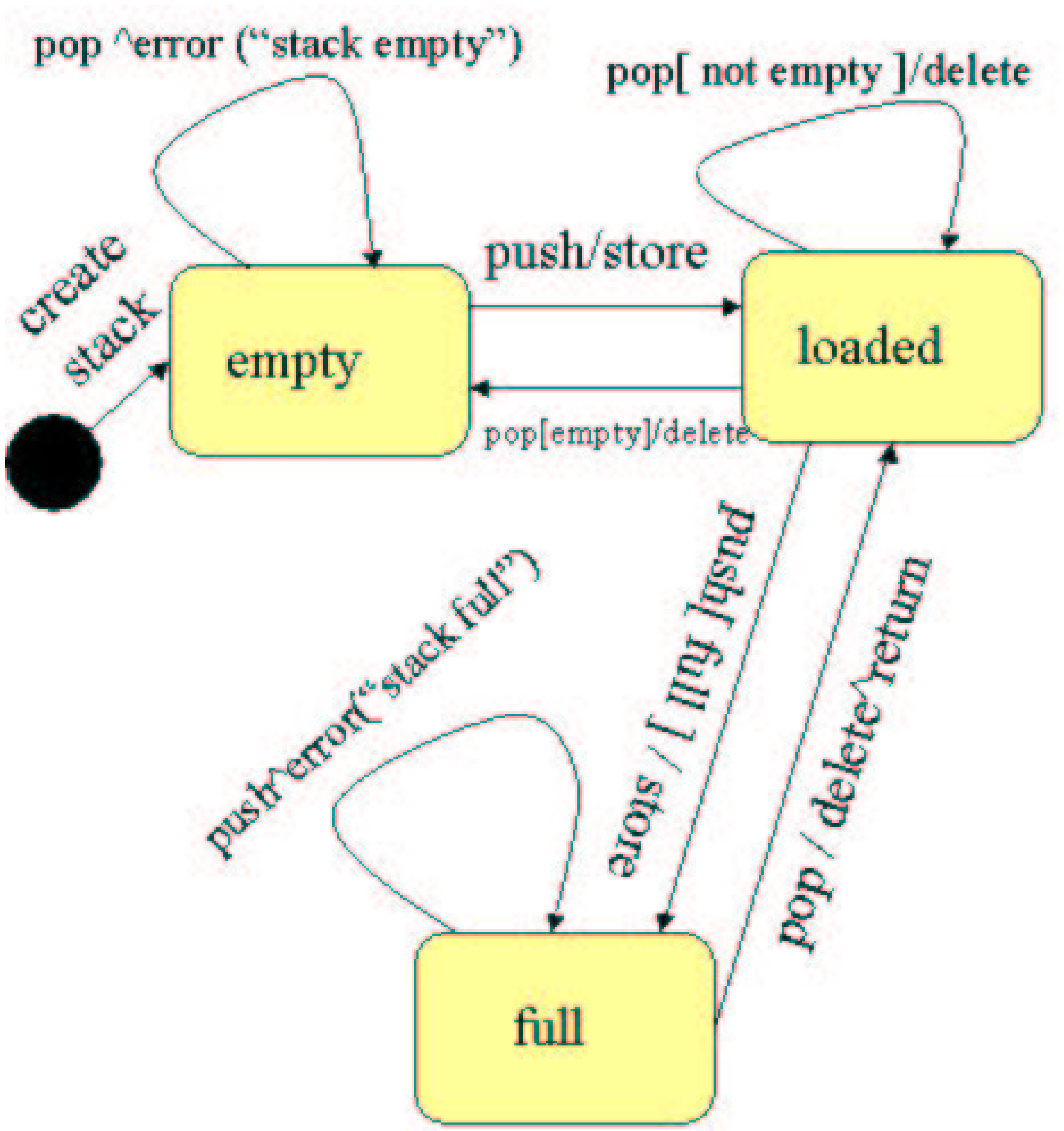

Figure 6.2: Representation of Statechart Semantics Using Stack

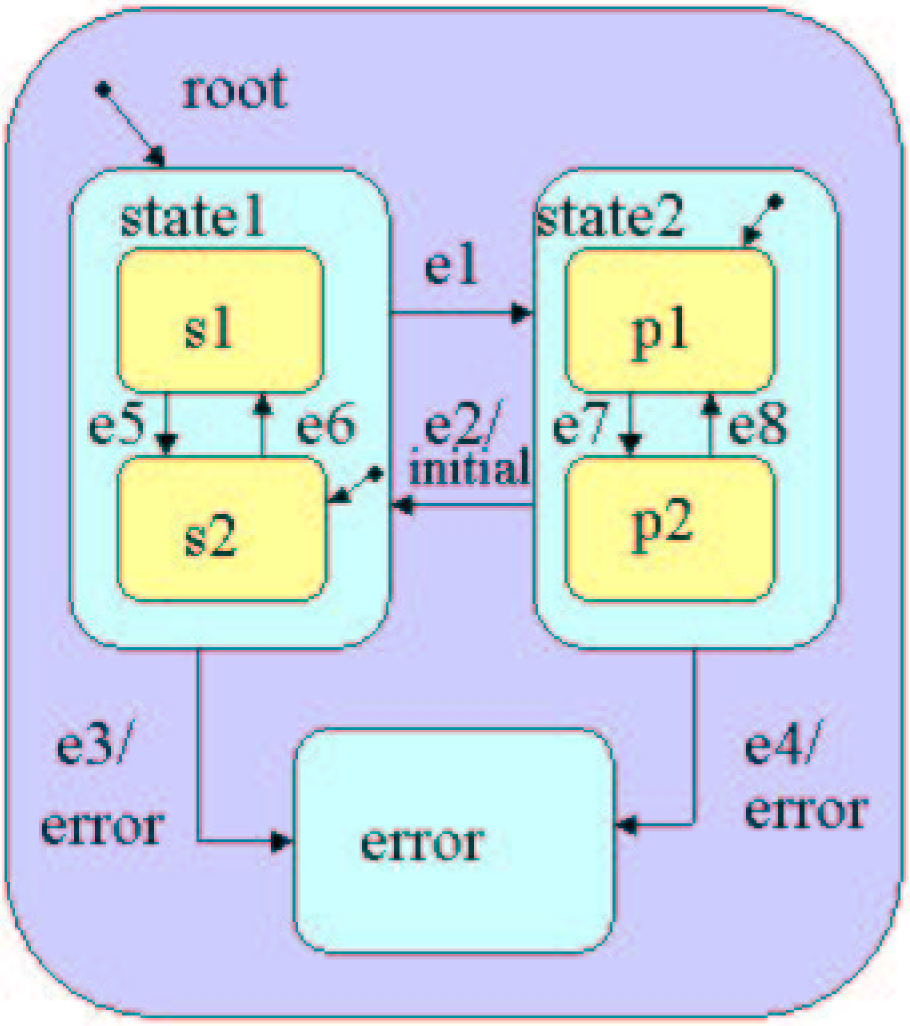

Figure 6.3: Representation of Generalization

6.6 Aggregation of States

State aggregation is the combination of one state from several other independent states. The composition is of a conjunctive type (composition of type 'and'), which implies that the object must simultaneously be in all states that constitute the aggregation. State aggregation corresponds to a kind of parallelism between state machines. In the figure 6.4, State **S** is an aggregation composed of two independent states **T** and **U**; **T** is composed of sub-states **X,Y** and **Z**, and **U** is composed of sub-states **A** and **B**. The domain of **S** is the Cartesian product of **T** and **U**.

An input transition into state **S** implies the simultaneous activation of state machines **T** and **U**, i.e, the object is initially placed in the composite state **(Z,A)**. When event **E3** occurs, the **T** and **U** state machines can keep evolving independently, which brings the object to the composite state **(X,A)**. State machines **T** and **U** may also evolve simultaneously, which is the case when event **E1** moves the object from composite state **(X,A)** to **(Y,B)**. Adding conditions to the transitions, such as the guard [**in Z**] placed on the transition from **B** to **A**, makes it possible to introduce dependency relationships between the components of the aggregate. When event **E4** occurs, the transition from **B** to **A** is only valid if the object is also in state **Z** at that time.

State aggregation, together with state generalization, simplifies the representation of state machines. Generalization simplifies by factorization, and aggregation simplifies by segmentation of the state space.

6.7 Role of Statecharts

- formally specify the behavior of objects
- increase the understanding of classes
- describe what happens when events occur within the system and it's environment
- provide abstract and partial descriptions of the actual code

In the next chapter, we discuss the various issues in the testing of statecharts.

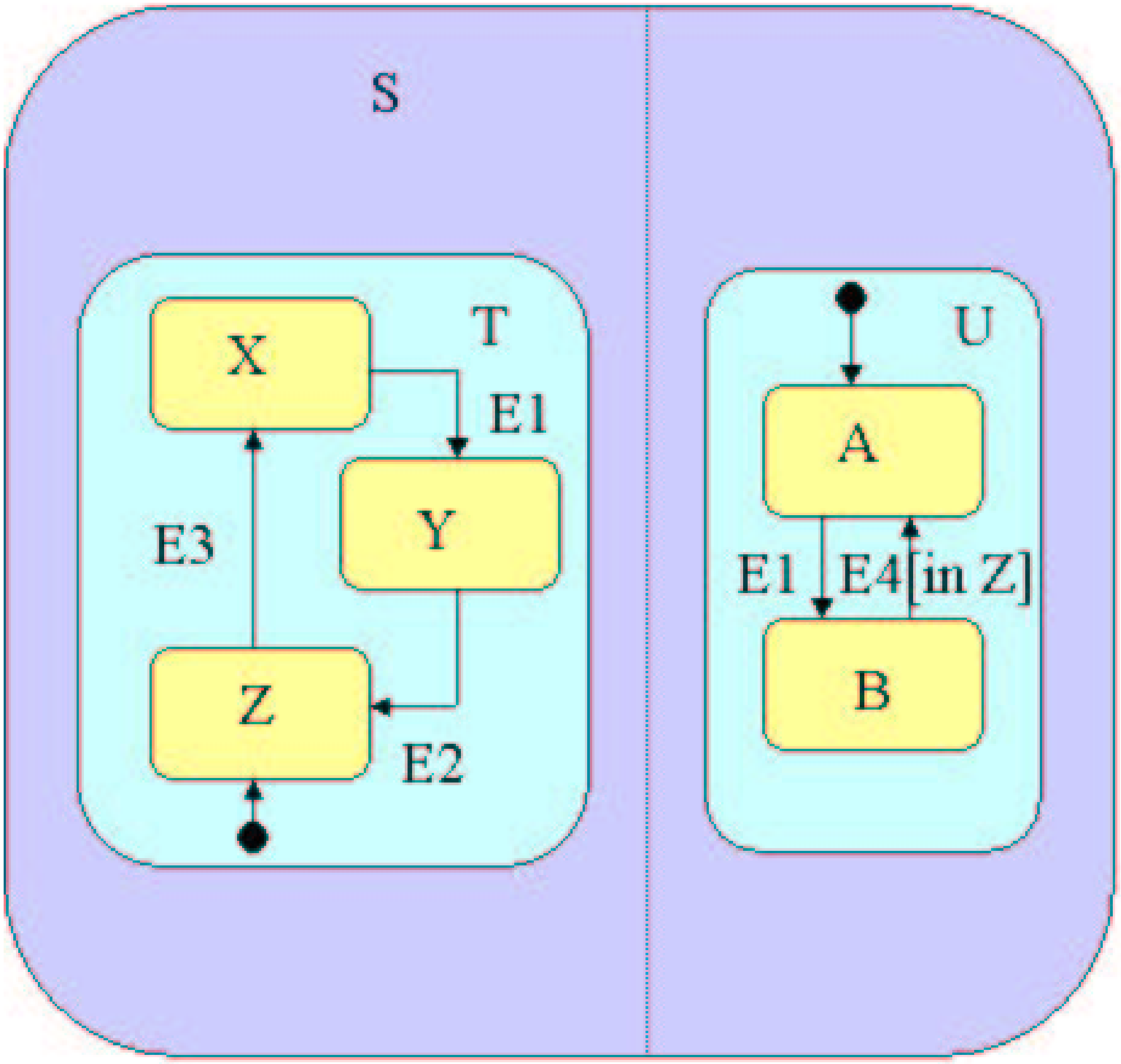

Figure 6.4: Representation of Aggregation of states

Chapter 7

Importance Of Testing Statecharts

7.1 Importance Of Testing Statecharts

Statecharts are becoming increasingly popular as a means of specifying safety-critical systems. It is therefore necessary to develop a systematic and rigorous approach to verifying that these systems conform to their specifications [5, 20, 8]. Unit testing is performed on safety- critical software to verify its functional behavior and as a means of obtaining the structural coverage metrics . Although commercially available tools exist to automate the collection of structural coverage metrics, unit testing still consumes a large proportion of total development costs [12]. Furthermore, the current approach to unit testing is often ad-hoc and the necessity to check the functional behavior of the system against its specification can be overshadowed by the more quantifiable targets of structural coverage. As a result, confidence in the quality of the software is compromised and the significance of unit testing within the context of demonstrating safety is reduced.

7.2 Methodology

Statecharts are a rich notation that allow complex system behavior to be specified in concise diagrams. This efficiency is made possible through a complicated syntax and semantics[3, 27, 13]. As a result, much of the system behavior is not immediately obvious from the diagrams. This poses a problem when designing automated functional testing techniques based on statechart specifications. They would require an understanding of the complex statechart semantics in order to interpret the behavior implied by the diagrams and generate tests based on this behavior. Notation specific techniques are then only needed to provide the translation between the graphical specifications and the intermediate notation[17, 4, 1].

Restricting the testing procedures to a particular domain (say, embedded safety-critical systems) increases the potential for automation. The proposed process is summarized in figure 7.1.

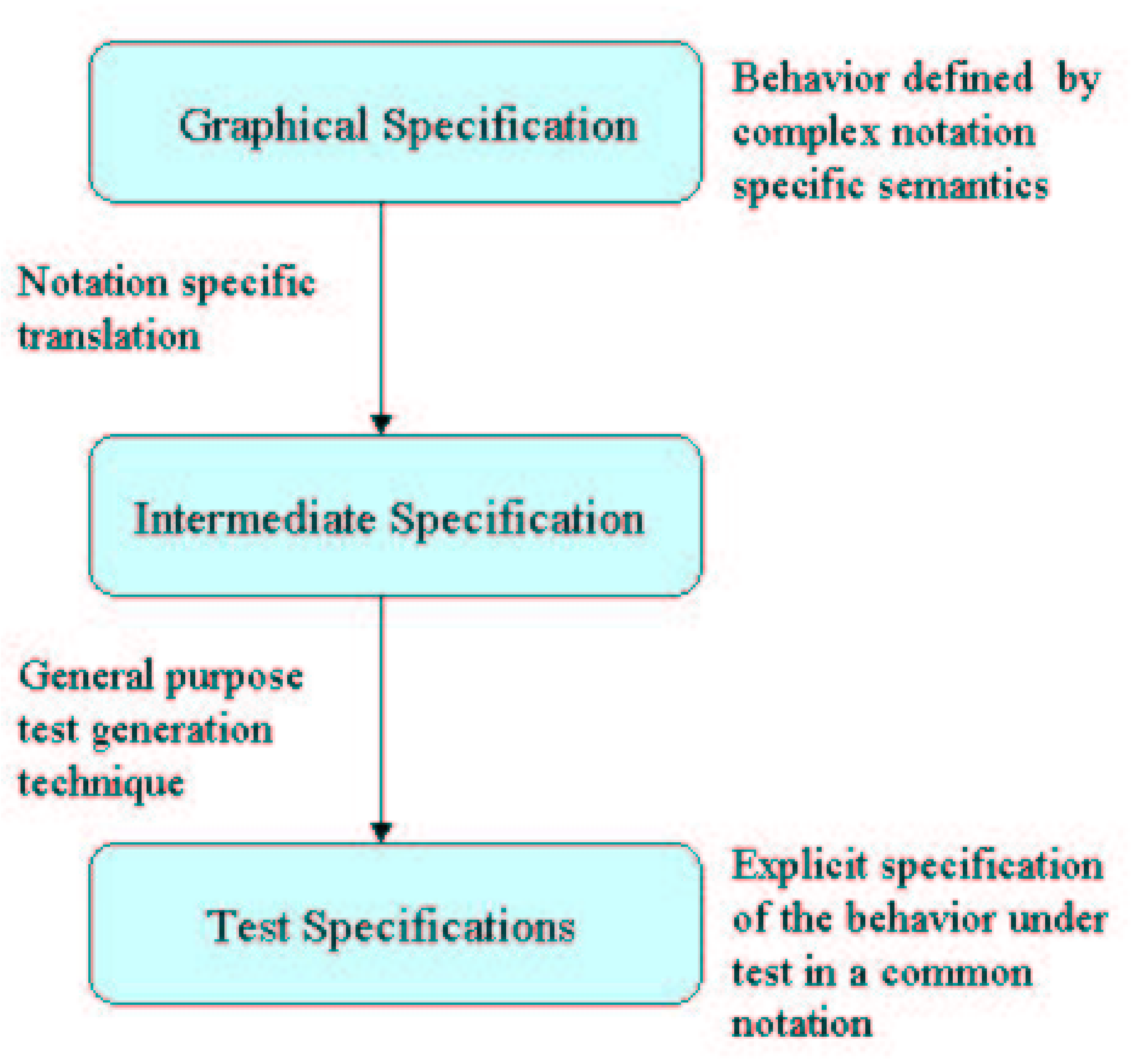
Graphical Specification
Behavior defined by complex notation specific semantics
Notation specific translation
Intermediate Specification
General purpose test generation technique
Test Specifications
Explicit specification of the behavior under test in a common notation

Figure 7.1: Test specifications from complex graphical notation

Statecharts allow complex reactive systems to be described in concise diagrams. This is made possible through the use of hierarchical state machines, an extended transition operation syntax and the ability to model concurrency[2, 23]. Each statechart has a state hierarchy. There is one state which is highest in the hierarchy and each state may contain a set of sub-states which are lower in the hierarchy than their surrounding super state[4]. States which contain no sub-states are basic states. Super states can be either AND-states or OR-states. When an OR-state is active, one and only one of its immediate sub-states is active at any time. The system moves between these sub-states sequentially according to the order in which the transitions connecting the states are triggered [14, 22, 26, 20].

Statecharts are a powerful visual formalism for capturing complex behavior, and apply well to both functionally decomposed systems and to object-oriented ones[9, 18, 21]. Objects are composite entities consisting both of information (attributes) and operations that act on that information (methods). State machines constrain the execution of those operations to better control and understand the object behavior. A state machine is defined by a set of existence conditions (states), event responses (transitions), and actions that are executed as we change states or take a transition. Statecharts add a number of useful extensions to the traditional flat Mealy-Moore state diagrams, such a nesting of states, conditional event responses via guards, orthogonal regions, and history. Although these extensions are mathematically equivalent to Mealy-Moore state machines, statecharts can be made much more parsimonious and vastly easier to understand, especially for complex state machines.

7.3 Coverage Metrics

Tests for a state machine or statechart can be designed from the functional specification of the state machine or statechart, independent of the structural design of the actual implementation. An obvious starting point is to test every transition. That is, to design test cases to set a current state, create the circumstances which lead to an event, to observe the action taken, the transition made, and the new state. As there is defined set of transitions in the state model, a *coverage* measure associated with this strategy is to measure the proportion of transitions exercised by a set of test cases.

Transition Coverage

Transition Coverage is defined as the ratio of the number of transitions exercised by the total number of transitions in the model. This measure is often referred to as **0-switch Coverage**, as described in[26]. Developments of this strategy are to test sequences of two transitions, three transitions, n transitions etc. For example, **2-Transition Coverage** is defined as the ratio of the number of sequences of two transitions exercised by the total number of sequences of two transitions in the state model. As the number of transitions exercised by each test case increases, the number of possible permutations and combinations of transitions within a sequence also increases, consequently increasing the number of test cases required to achieve the associated coverage metrics. Increased complexity consequently makes achievement of coverage for long sequences of transitions impractical[19]. As a minimum each test case should be specified as:

an initial "current state", inputs to the software, the expected action(if any), including a transition(if any) and outputs(if any), the resulting "next state".

This gets more complicated when test cases are designed to execute sequence of two or more transitions. For such test cases, the specification of the test cases should include the sequence of events, actions, transitions and states involved in the test case.

Any strategy aimed at testing specified transitions is biased towards **positive testing**. That is, test cases are designed to exercise that software does what it is supposed to do. A thorough test should also include **negative testing**, to verify that the software does not do things that it is not supposed to do. Within existing positive test cases, checks should be incorporated to ensure that there have been no unwanted side effects. Furthermore, additional test cases should be designed to ensure that invalid actions and transitions cannot be induced.

State Coverage

The *State coverage* is defined as the ratio of the number of states covered in the test cases by the total number of states in the given model.

Event Coverage

The *Event coverage* is defined as the ratio of the number of events covered in the test cases by the total number of events in the given model.

State-Event Coverage

Any strategy for ensuring that negative tests are designed to verify freedom from such bugs is to design test cases which subject each state of the state machine to each event in the total set of events, not just the legal events for that state. The *State-Event Coverage* is defined as the ratio of the number of state-event pairs exercised by the product of the number of states and the number of events in the given model. If the state machine is *fully specified*, such that in each state, for each event, a unique transition was defined (in some cases, null transitions), then Transition-Coverage or 0-Switch Coverage would be the sane as State-Event Coverage. In practice, it is common for state machines to not be *fully specified.*

7.4 Summary

In this chapter, we have presented the importance of testing statecharts and coverage metrics. In the next chapter, we will discuss the design specification of the StateTest tool.

Chapter 8

Design Specification Of StateTest Tool

8.1 StateTest : A New Tool For Testing Statecharts

The StateTest tool is used to test the Statecharts behavioral description. It takes the Statechart specification and the test script as the *input* and gives the error report as the *output*.

8.2 Inputs and Outputs To StateTest Tool

The system has two input files and produces three types of outputs. This can be viewed from the figure 8.1.

Inputfile1: Contains the list of Statechart specification. The format of the file is:

```
Initial_State;Event;Next_State;Actual_Action;
```

Where all fields are combination of both characters and digits. An example of this file for the figure 6.3 is :

```
state1;e1;state2; (no action part)
state2;e2;state1;initial; (action part present)
s1;e5;s1; (no action part)
```

Inputfile2: Contains the list of test scripts. The format of the file is :

```
State;Sequence_Of_Events;Expected_State;Expected_Action;
```

Where all fields are combination of both characters and digits. An example of this file is the figure in 6.3 is :

```
state1;e1,e2,e6,e5,e6,e1,e7,e8,e7,e4;error;error; (action part present)
s1;e1,e2;s1;initial; (action part present)
s2;e1,e7,e8;p1; (no action part)
```

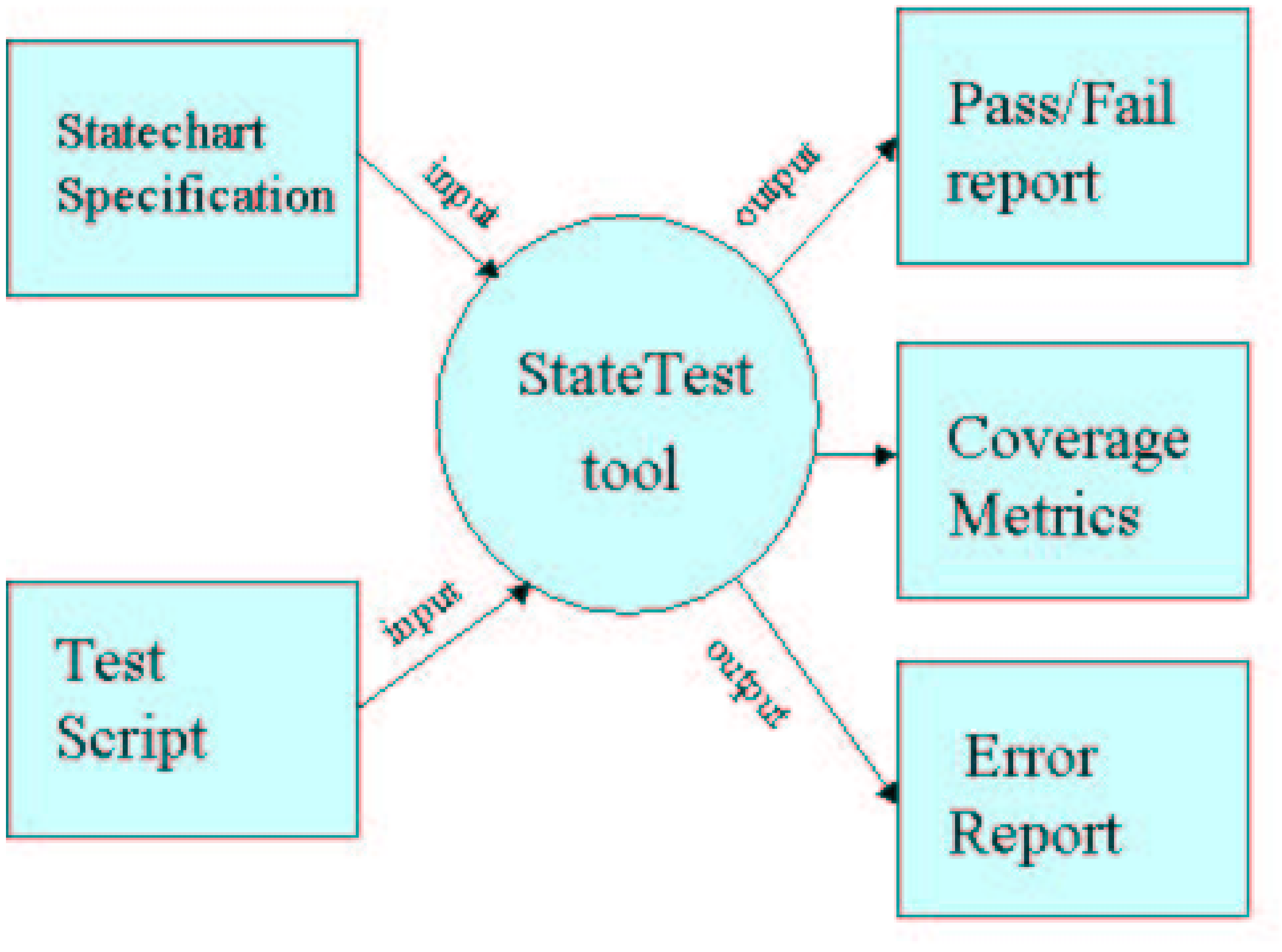

Figure 8.1: Level 0 DFD for StateTest tool

Output1: For each test case, it gives pass or fail report and finally it gives information about the number test cases are performed among them number of test cases are failed and number of test cases are passed.

Output2: It gives the coverage metric information, i.e, State Coverage, Event Coverage, State-Event Coverage and Transition Coverage based on the test cases performed.

Output3: It gives error messages. For example, input file does not exist, inputfile1 has error, inputfile2 has error, and the given test case is not in the statechart specification.

User Interface: Only one user command is required. The file names can be specified in the command line itself or the the system prompts for the input file names.

8.3 The Skeleton Algorithm for Testing Phase

The overall view of the StateTest tool can be seen from the figure 8.2.

The algorithm for the testing phase involved in the StateTest tool is :

```
# Skelton Algorithm for testing the statechart specification
# Test Case' Format :
# state;sequence_of_events;expected_state;expected_action;
# Specification of statechart Format
# initial_state;event;next_state;actual_action;
# tevent means event in the test case
# event means event in the specification file of statechart
# " + " means concatenation of  strings
# " $" here used for default state indicating that no transition at all

read testscript file;
take one test case at a time from the testscript file;
read state
read tevent
read expected_state
read expected_action

event_count:= count(tevent);

#check if event field has more than one event

if(event_count > 1)

        while(event_count >= 1)
        {
                temp_state_pair := state + event1 ;

                read specification file;
                take one line at a time;
                actual_state_pair := initial_state + event;
                compare temp_state_pair and actual_state_pair;

                if(both are equal)
                {

                        state := next_state;
                        temporary_action := actual_action;
                }

                elseif(tevent == event)
                {
                 if(state contains intial_state || intial_state contains state)
                        {
                                if((intial_state == default_state) ||
                                {
                                        (state == default_state) &&
                                        (next_state != default_state))

                                        state := next_state;
                                        temporary_action := actaul_action;
                                }

                                elseif((state != deafult_state) &&
                                        ((next_state != default_state) ||
                                        (expected_state != next_state)))
                                {

                                        state := next_state;
                                        temporary_action := actual_action;
                                }
                        }

                }

                else
                {
                        exit;
                }

        read the statechart specification file till ending;

        }

event_count := event_count - 1;
```

```
continue the process until event_count == 0;

# to get pass/fail report

        temp_state_pair := state + tevent;
        acual_state_pair := intial_state + event;
        expected_nextstate_action_pair := expected_state + expected_action;

        actual_nextstate_action_pair := next_state + actual_action;
        pass_count := 0;
        fail_coubt:= 0;

if(event_count == 1)

                #concate state with tevent
                temp_state_pair := state + tevent;

                #concate intial_state with event
                acual_state_pair := intial_state + event;

       if(temp_state_pair == actaul_state_pair)
           {
             if(expected_nextstate_action_pair == actual_nextstate_action_pair)
                               {
                                       pass_count := pass_count + 1;
                               }

             else
              {
                if((expected_state == default_state) &&
                    (expected_action == actual_action))

                       {
                               pass_count := pass_count + 1;

                       }
                else
                       {
                               fail_count := fail_count + 1;
                       }
               }
             }

             else
               {
                 if(tevent == event)
```

```
                    {
                      if((state contains intial_state) ||
                         (intial_state contains state))
                          {
                            if(expect_nextstate_action_pair ==
                                actaul_nextstate_action_pair)
                              {
                                 pass_count := pass_count + 1;

                              }
                             else
                              {
                               if((expected_state contains next_state)||
                                   (next_state contains expected_state))
                                   if((expected_state == default_state)
                                      &&(expected_action == actaul_action))
                                        {
                                             pass_count := pass_count + 1;
                                        }
                                     else
                                        {
                                             fail_count := fail_count + 1;
                                        }
                                }
                              }
                          }

#to handle the innermost states of the hierarchy

                    else
                     {

                     {
                      if((intial_state contains any one of the transitions)
                         ||(any one of the transition contains intial_state)
                              &&(expected_action == actual_action))
                              {
                                      pass_count := pass_count + 1;
                              }
                              else
                              {
                                      fail_count := fail_count + 1;
                              }
                      }
              }
        }
```

```
# to handle the situation having more than one event

elseif(event_count > 1)
        {
                if(state == expected_state)
                {
                        if(expected_action == actual_action)

                                pass_count := pass_count + 1;
                        }
                        else
                        {
                                fail_count := fail_count + 1;
                        }
                else
                {

                    if((expected_state contains state)||
                       (state contains expected_state))
                    {

                          if(((state == default_state)&&
                             (expected_state == next_state))
                               ||((expected_state == default_state)
                                  &&(expected_action == actual_action)))
                               {

                                       pass_count := pass_count + 1;
                               }





                        else
                        {
                                fail_count := fail_count + 1;
                        }
                else
                {

                   if((expected_state contains state)||
                      (state contains expected_state))
                    {

                      if(((state == default_state)&&
                        (expected_state == next_state))
                        ||((expected_state == default_state)
                        &&(expected_action == actual_action)))
                               {

                                       pass_count := pass_count + 1;
                               }
                               else

                               {
                                       fail_count := fail_count + 1;
                               }
                  }
               }

#to handle default_states

        if(expected_state == "$")
        {
                if(temp_state_pair != actual_state_par)
                        {
                                pass_count := pass_count + 1;
                        }
                else
                        {
                                fail_count := fail_count + 1;
                        }
        }


continue the reading of the test script file till ending;

  }
```

The overall view of the modules involved in this process are as shown in the figure 8.3.

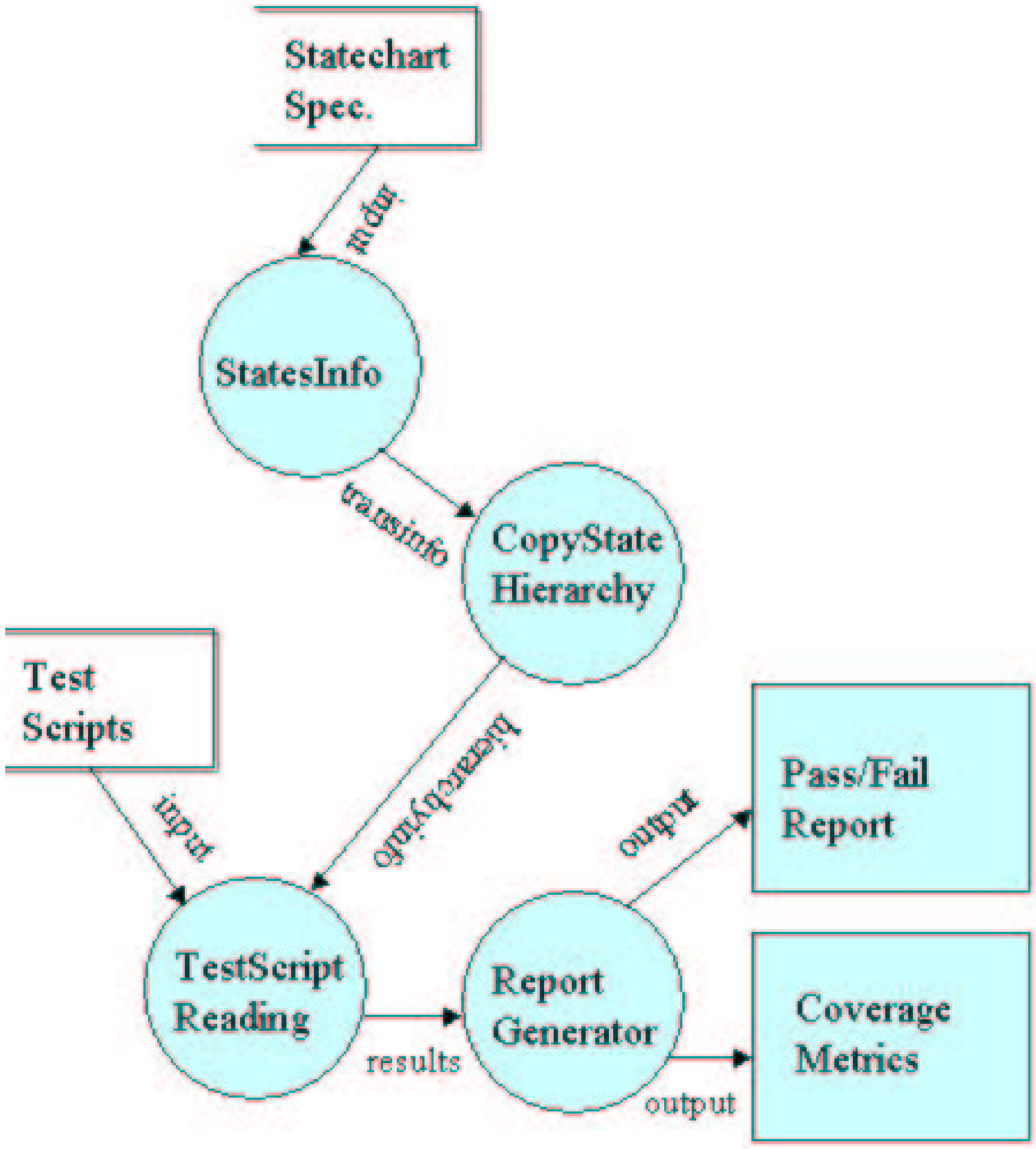

Figure 8.2: Level 1 DFD for StateTest Tool

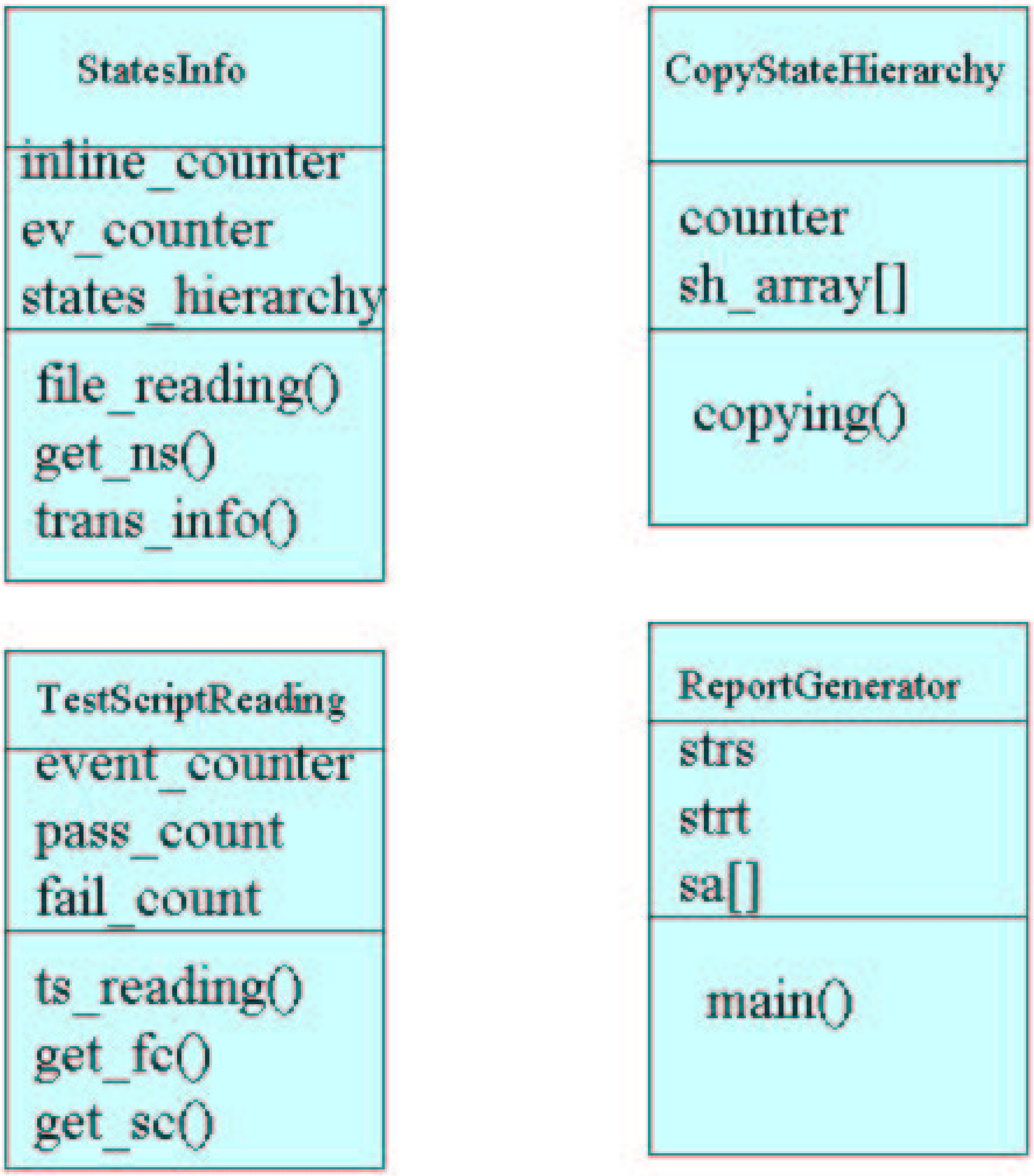

Figure 8.3: Initial Object Design for the StateTest tool

8.4 Components Of StateTest Tool

The modules involved in this StateTest tool are :

- TestScriptReading
- StatesInfo
- CopyStateHierarchy
- ReportGenerator

TestScriptReading

The TestScriptReading module does the following:

- reads the test script file
- takes one test case at a time
- invokes the CopyStateHierarchy module to get the required statechart specification information
- final report is given to the ReportGenerator

StatesInfo

The StatesInfo module does the following :

- reads the Statechart Specification file
- gets the information about hierarchy, transitions, state-key pair,and the number of states (including default states)
- passes these results to the ReportGenerator
- passes hierarchy information to the CopyStateHierarchy module

CopyStateHierarchy

This module does the following :

- gets the states' hierarchy information from the StatesInfo module
- gives the hierarchy information to the TestScriptReading module

ReportGenerator

After getting results from the TestScriptReading module, it does the following:

- generates Pass/Fail Report
- generates coverage metrics

The overall view of the interaction among all these modules is given by the figure 8.4.

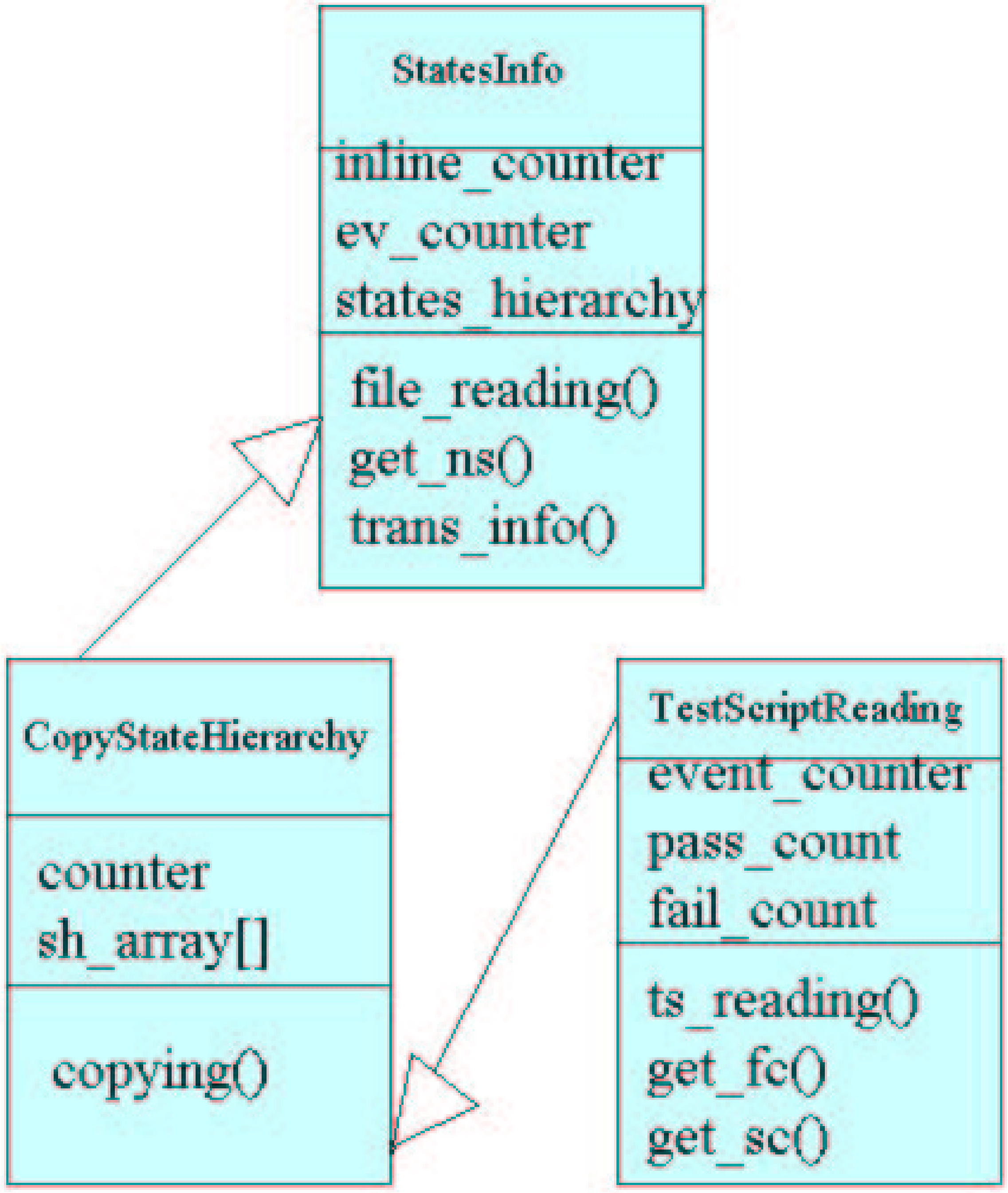

Figure 8.4: Detailed Object Design for the StateTest Tool

8.5 Detailed Design Specification Of StateTest Tool

We have implemented StateTest using the Java programming language. We have used the four classes, whose abstract description is given below.

```
// This is the specification for the Object-Oriented Design for the StateTest tool.
// It gives all the major classes and most of the major data members and operations
// for these classes. The major parameters of the various operations are also
// given, which shows how objects are being made visible. All the major object
// declarations are also given.

class ReportGenerator
{

        // files reading by prompting

        BufferedReader input = new BufferedReader(
                new InputStreamReader(System.in));
                String  strt = input.readLine();
                String strs = input.readLine();

//local variables declarations

                int nstates;
                int ntrans;
                int tevents;
                int dstates;
                int tsize2;
                long start ;
                long end ;
                double total_events = tevents;
                double event_coverage ;
                double total_transitions ;
                float transitions_exercised ;
                double transition_coverage ;
                double total_states ;
                double state_coverage ;
                double  prod_events_states ;
                double StateEvent_Coverage;
//Objects creation and method invocation

        StatesInfo fob = new StatesInfo();
                fob.file_reading(strs);

                nstates = fob.get_ns();
                ntrans = fob.get_nt();
                tevents = fob.get_te();
                dstates = fob.get_ds();
                tsize2 = fob.get_tsize();
                String  sa[]= new String[nstates];
                sa = fob.sarray_info();
                String ta[] = new String[ntrans];
                ta = fob.trans_info();
                String sh2[][] = new String[nstates][2*nstates];
        CopyStateHierarchy cob = new CopyStateHierarchy();
                sh2 = cob.copying(nstates,sa);
        TestScriptReading tso = new TestScriptReading();
                tso.ts_reading(tsize2,ta,strt,sh2);

// method invocation
                state_counter=tso.get_sc();
                event_counter = tso.get_ec();
                state_event_counter = tso.get_sec();
                transition_counter = tso.get_tc();
                start = tso.get_stime();
                end = tso.get_etime();
                pass_count = tso.get_pc();
                fail_count = tso.get_fc();
}
class StatesInfo
{
        //attributes of the class
                String strt ;
                String strs ;
                int ns;
                int nt;
                int ds;
                int te;
                int inline_counter;
                public String sh_array[];
                public String states_hierarchy[][];
                static  Vector states_file = new Vector();
                Vector transition_file = new Vector();
                Hashtable state_key_pair = new Hashtable();
                Enumeration states_keys;
                HashSet ev_counter = new HashSet();
//methods in the class StatesInfo

                public void file_reading(String);
                public int get_ns();
                public int get_nt();
                public int get_te();
                public int get_ds();
                public  int get_tsize();
                public static String[] sarray_info();
                public String[] trans_info();

//methods variables:
                String fin;
                String num_states ;
                String num_transitions ;
                String num_default_states ;
                String num_events ;
                String dummy_states ;
                String dummy_transitions ;
                int ecount;
                int entry_counter;
}
public class CopyStateHierarchy extends StatesInfo
{
//attributes of the class CopyStateHierarchy
                String sh_array[]= new String[ns];
                sh_array = s1;
                String  states_hierarchy[][] = new String[ns][2*ns];
                int counter;
                int x ;
                int y ;
//method to copy the hierarchical information of the states

                public String[][] copying(int z, String s1[])
}
public class TestScriptReading extends CopyStateHierarchy
{
        //attributes of the class TestScriptReading
                int pass_count ;
                int fail_count ;
                int event_count ;
                int event_count1 ;
                int event_count2 ;
                int testcase_count ;
                long start ;
int field_count;
                long end ;
                long total_time;
                int event_counter ;
                int entry_counter ;
                int state_counter ;
                int state_event_counter ;
                int transition_counter ;
                int tsize ;
                String transition_array[] = new String[int];
                String strt ;
        // flags declaration

                boolean TestFlag = false;
                boolean dsflag = false;
                boolean ds_entry_flag = false;
                boolean hlevel_flag = false;
                boolean hlevel_flag1 = false;
        //Hash sets

                HashSet states = new HashSet();
                HashSet eventsc = new HashSet();
                HashSet states_events = new HashSet();
                HashSet transitions = new HashSet();
                HashSet dnext_states = new HashSet();
                HashSet dsnext_states = new HashSet();
                HashSet def_states = new HashSet();
                HashSet defnext_states = new HashSet();

//methods declarations

        public void ts_reading(int g1, String tr1[], String z1, String sh1[][]);
                public int get_ec();
                public int get_tc();
                public long get_stime();
                public long get_etime();
                public double get_sec();
                public int get_pc();
                public int get_fc();
}
```

8.6 Explanation Of the Classes

The project consists of mainly four classes namely, *StatesInfo, CopyStateHierarchy, TestScriptReading, and ReportGenerator*. The StatesInfo is used to get required information from the input. It extracts the information about number of states, number of transitions in the given input. CopyStateHierarchy is used for the copying hierarchical information of the states. The class TestScrptReading is used for performing test cases. ReportGenerator is used to initialize the objects and display the result.

In the next chapter, we describe the experiments performed using StateTest and discuss the results.

Chapter 9

Simulation Results

9.1 Statechart Specification File

The StateTest tool takes the statechart specification file as an **input** n the form generated by the **Statechart Editior (SCE)** tool, which was developed by a group in the **BARC (Bhabha Atomic Research Centre)**. For example, for the figure 6.3, the code generated by the **Statechart Editior** is :

```
statechart
11 11 11 11

1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1

0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 root 0 4 0 0 0 0 0 0 2000 2000 0 1110 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 state1 0 4 62 61 0 0 0 0 293 255 0 1110 0 0 0 0 0 0 0 0 0 0 0
2 0 0 0 0 state2 0 4 367 59 0 0 0 0 535 255 0 1110 0 0 0 0 0 0 0 0 0 0 0
3 0 0 0 0 error 0 4 242 301 0 0 0 0 387 376 0 1110 0 0 0 0 0 0 0 0 0 0 0
4 0 0 0 0  0 1 323 45 0 30 0 0 0 0 0 0 0 0 0 0 0
5 0 0 0 0 s1 0 4 110 93 0 0 0 0 185 147 0 1110 0 0 0 0 0 0 0 0 0 0 0
6 0 0 0 0 s2 0 4 166 181 0 0 0 0 235 229 0 1110 0 0 0 0 0 0 0 0 0 0 0
7 0 0 0 0 p1 0 4 407 87 0 0 0 0 485 143 0 1110 0 0 0 0 0 0 0 0 0 0 0
```

```
8 0 0 0 0  0 1 464 171 0 30 0 0 0 0 0 0 0 0 0 0 0
9 0 0 0 0  0 1 234 155 0 30 0 0 0 0 0 0 0 0 0 0 0
10 0 0 0 0 p2 0 4 399 195 0 0 0 0 483 240 0 1110 0 0 0 0 0 0 0 0 0 0 0

4 30 1 1110 0  0 0 0 0 4 323 45 292 73 0
8 30 7 1110 0  0 0 0 0 4 464 171 461 142 0
9 30 6 1110 0  0 0 0 0 4 234 155 234 181 0
5 1110 6 1110 8 e5 0 0 0 0 4 110 143 166 208 0
6 1110 5 1110 8 e6 0 0 0 0 4 199 181 185 135 0
1 1110 3 1110 8 e3/error 0 0 0 0 4 262 255 305 301 0
2 1110 3 1110 8 e4/error 0 0 0 0 4 391 255 360 301 0
7 1110 10 1110 5 e7  0 0 0 0 4 407 136 422 195 0
10 1110 7 1110 5 e8  0 0 0 0 4 449 195 443 143 0
1 1110 2 1110 9 e1 0 0 0 0 4 293 143 367 150 0
2 1110 1 1110 9 e2/initial 0 0 0 0 4 367 187 293 192 0
```

In this representation many fields are not required for the StateTest tool. It takes this form as an input, and internally generates the specification file as a four tuple, as was seen in the previous chapter 8. The first line is just name 'statechart', the next line containing four 11's, first two 11's indicates the number of states and number of transitions (including default states) in the given model, remaining two 11's are simply copies of the first two 11's. The next two consecutive lines containing 1's indicating the number of states and number transitions present. The next block represent the hierarchical state information. It is of rectangular matrix of 11 by 22 (2*11). For each each state, there are two columns reserved, one is for indicating **normal states** and another for indicating **orthogonal states**. The next block meant for **state numbering**. The last block indicates **transitions** among the states involved in the given model.

9.2 Statechart Test Cases File

The test cases for the example statechart fig 6.3, are as follows :

```
#Specification file name :example_figure_specification.txt
#Test case file_name : example_figure_testcases.txt
#Test Case' Format :
# State;Sequence_Of_Events;Expected_State;Expected_Action;

1;e1,e2,e6,e5,e6,e1,e7,e8,e7,e4;3;error;
1;e6,e5;6;
5;e1,e2;5;initial;
1;e1,e2,e5;6;
5;e1,e7;10;
6;e1,e7,e8;7;
```

9.3 Demo Results

The demo results for the given Statechart specification and Test cases are as follows:

```
Enter the Test Cases file name : example_figure_specification.txt
Enter the Statechart Specification file name : example_figure_testcases.txt
Number of states : 11
Number of transitions:11
Inline counter value in reading State Info.is : 1
Inline counter value in reading State Hierarchy Info. is :11
Inline counter value in reading State - Key pair is :11
Key : State
9:0
8:0
7:p1
6:s2
5:s1
4:0
3:error
2:state2
10:p2
1:state1
0:root
Inline counter value in reading Transition Info. is :11
default states:3
total events:8
ns:11
nt:11
te:8
ds:3

field count :4

Test Case : 1
1;e1,e2,e6,e5,e6,e1,e7,e8,e7,e4;3;error;
expected state: 3
expected action:error

event(s) are: e1,e2,e6,e5,e6,e1,e7,e8,e7,e4

The number of events to be parsed are: 10
Temporary Event : e1
Temporary State_Event Pair : 1e1
```

```
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;

1;e3;3;error;
 2;e4;3;error;
 7;e7;10;
 10;e8;7;
 1;e1;2;
Temporary State : 2
Temporary Action: Temporary Event : e2
Temporary State_Event Pair : 2e2
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
 1;e3;3;error;
 2;e4;3;error;
 7;e7;10;
 10;e8;7;
 1;e1;2;
 2;e2;1;initial;
Temporary State : 1
Temporary Action:initial
Temporary Event : e6
Temporary State_Event Pair : 1e6
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
Temporary Event : e5
Temporary State_Event Pair : 5e5
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
Temporary State : 6
Temporary Action:
Temporary Event : e6
Temporary State_Event Pair : 6e6
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
Temporary State : 5
Temporary Action:
Temporary Event : e1
Temporary State_Event Pair : 5e1
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
 1;e3;3;error;
 2;e4;3;error;
 7;e7;10;
 10;e8;7;
 1;e1;2;
Temporary Event : e7
Temporary State_Event Pair : 2e7
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;

 6;e6;5;
 1;e3;3;error;
 2;e4;3;error;
 7;e7;10;
Temporary Event : e8
Temporary State_Event Pair : 10e8
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
 1;e3;3;error;
 2;e4;3;error;
 7;e7;10;
 10;e8;7;
Temporary State : 7
Temporary Action:
Temporary Event : e7
Temporary State_Event Pair : 7e7
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
 1;e3;3;error;
 2;e4;3;error;
 7;e7;10;
Temporary State : 10
Temporary Action:
Temporary Event : e4
Temporary State_Event Pair : 10e4
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
 1;e3;3;error;
 2;e4;3;error;
 test cases's state: 3
 test case's event:e1,e2,e6,e5,e6,e1,e7,e8,e7,e4
 test cases's action: error
  Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 loop is in the super state spec. seq_events
 this test case is passed

field count :3

Test Case : 2

1;e6,e5;6;
 expected state: 6
 event(s) are: e6,e5
The number of events to be parsed are: 2
Temporary Event : e6
Temporary State_Event Pair : 1e6
 Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
Temporary Event : e5
Temporary State_Event Pair : 5e5
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
Temporary State : 6
Temporary Action:
 test cases's state: 6
 test case's event:e6,e5
 test cases's action: error
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
loop is in the super state spec. seq_events
 this test case is passed

field count :4

Test Case : 3
 5;e1,e2;5;initial;
 expected state: 5
 expected action:initial
event(s) are: e1,e2
The number of events to be parsed are: 2
Temporary Event : e1
Temporary State_Event Pair : 5e1
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
 1;e3;3;error;
 2;e4;3;error;
 7;e7;10;
 10;e8;7;
 1;e1;2;
Temporary Event : e2
Temporary State_Event Pair : 2e2
```

```
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
 1;e3;3;error;
 2;e4;3;error;
 7;e7;10;
 10;e8;7;
 1;e1;2;
 2;e2;1;initial;
Temporary State : 1
Temporary Action:initial
test cases's state: 1
test case's event:e1,e2
test cases's action: initial
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
loop is in the super state spec.seq_events
 this test case is failed

 field count :3

 Test Case : 4
 1;e1,e2,e5;6;
expected state: 6
 event(s) are: e1,e2,e5
 The number of events to be parsed are: 3
Temporary Event : e1
Temporary State_Event Pair : 1e1
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
 1;e3;3;error;
 2;e4;3;error;
 7;e7;10;
 10;e8;7;
1;e1;2;
Temporary State : 2
Temporary Action:
```

```
Temporary Event : e2
Temporary State_Event Pair : 2e2
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
 1;e3;3;error;
 2;e4;3;error;
 7;e7;10;
 10;e8;7;
 1;e1;2;
 2;e2;1;initial;
Temporary State : 1
Temporary Action:initial
Temporary Event : e5
Temporary State_Event Pair : 1e5
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
 1;e3;3;error;
 2;e4;3;error;
 7;e7;10;
 10;e8;7;
 1;e1;2;
 2;e2;1;initial;
 test cases's state: 1
 test case's event:e1,e2,e5
 test cases's action: initial
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
loop is in the super state spec.seq_events
 this test case is failed

field count :3

Test Case : 5
5;e1,e7;10;
 expected state: 10
 event(s) are: e1,e7
```

```
The number of events to be parsed are: 2
Temporary Event : e1
Temporary State_Event Pair : 5e1
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
 1;e3;3;error;
 2;e4;3;error;
 7;e7;10;
 10;e8;7;
 1;e1;2;
Temporary Event : e7
Temporary State_Event Pair : 2e7
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
 1;e3;3;error;
 2;e4;3;error;
 7;e7;10;
test cases's state: 10
test case's event:e1,e7
test cases's action: initial
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
loop is in the super state spec. seq_events
 this test case is passed

field count :3

Test Case : 6
 6;e1,e7,e8;7;
 expected state: 7
event(s) are: e1,e7,e8
 The number of events to be parsed are: 3
Temporary Event : e1
Temporary State_Event Pair : 6e1
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
 1;e3;3;error;
 2;e4;3;error;
 7;e7;10;
 10;e8;7;
 1;e1;2;
Temporary Event : e7
Temporary State_Event Pair : 2e7
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
 1;e3;3;error;
 2;e4;3;error;
 7;e7;10;
Temporary Event : e8
Temporary State_Event Pair : 10e8
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
 6;e6;5;
 1;e3;3;error;
 2;e4;3;error;
 7;e7;10;
 10;e8;7;
Temporary State : 7
Temporary Action:
test cases's state: 7
test case's event:e1,e7,e8
test cases's action: initial
Statechart Specification Checking is going on
 4;1;
 8;7;
 9;6;
 5;e5;6;
loop is in the super state spec. seq_events
 this test case is passed
Number of test cases performed are :  6
Number of test cases passed are :   4
Number of test cases failed are :   2
Number of states covered are :   7
Number of states  are :   7.0
State Coverage is    100%

Number of events  are :   8.0
Number of events covered  are :   7
Event Coverage is    87%

Number of states and events covered  are :   12.0
Number of states and events pair  are :   56.0
StateEvent Coverage is    21%

Number of transitions exercised are :   7
Number of transitions are :   8.0
Transition Coverage is :87%
Total time  for 6 test cases is :630ms
Time  for  One test case is :105 ms
```

9.4 Summary Of the Results

From the getting results, it is obsevred that the number of test cases performed are 6 of which 4 are passed and 2 are failed. The state coverage is 100% that indicates given test cases covered all the states in the given model. It is also interesting to see that **transition coverage** and **Event coverage** are the same. Because, for the given Statechart model, i.e, figure 6.3, there is a *distinct transition* for each *distinct event*. The State-Event coverage is low, because given test cases had covered less number of state-evet pairs (12) when compare to the acutally existed event-pairs in the given model are 56. Generally, this metric is meant for **negative testing**. It is observed that time taken for a test case is reasonable, i.e, simply 105ms. Observe that the given test cases are having sequence of events also.

Chapter 10

Conclusions

In this project, we first performed a survey of software testing techniques, studied the requirements for testing statecharts. Subsequently, we studied some commercially available software testing tools. Finally, we designed **StateTest**, a tool for testing statecharts, implemented StateTest in Java and have performed experiments with it.

10.1 Future Work

StateTest presently tests the Statechart specification that has *hierarchy.* This hierarchy can be many levels. The code generated by the **StateTest Editor(SCE)** is very specific and reserved one column for each state to indicate the *orthogonal* state presence. So using that field, one can easily extends the functionality of the StateTest tool to test the Statechart specification having *orthogonality* and *history* information. Note that code generated by the Statechart Editor from the given model of Statechart, is one of the *input* to the StateTest tool. That too, in the *transition block*, there is *numbering* for each state, that indicates, whether the state is normal, orthogonal, default, or history. So, this may be used to handle orthogonal states and history states.

StateTest can also be extended and generalized to test statecharts having different specification formats.

Bibliography

[1] A. Peron A. Maggiolo-Schettini. A graph rewriting framework for statechart semantics. In *Proceedings of Fifth International Workshop on Graph Grammers and Their Application to Computer Science, Springer LNCS vol 1073, pp 106–121*, 1996.

[2] A. Peron A. Maggiolo-Schettini. Retiming techniques for statecharts. In *Proceedings of FTRTFT'96, Springer LNCS vol 1135, pp 55–71*, 1996.

[3] M. Merro A. Maggiolo-Schettini. Priorities in statecharts. In *Proceedings of LOMAPS'96, Springer LNCS vol 1192, pp 404–429*, 1997.

[4] S. Tini A. Maggiolo-Schettini, A. Peron. Equivalences of statecharts. In *Proceedings of CONCUR'96, Springer LNCS vol 1119, pp 687–702*, 1996.

[5] Jie Pan A.Jefferson Offutt, Zhenyi Jin. The dynamic domain reduction procedure for test data generation. In *Software-Practice and Theory*, 1999.

[6] Sinan Si Alhir. *UML in a Nutshell : A Desktop Quick Reference.* O'Reilly & Associates,Inc.,, 1998.

[7] Rumbaugh Booch and Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, Reading, MA, 1998.

[8] Moshe Cohen. *Scoping and Testing in State Based Models.* i-Logix,Inc., 223rd Avenue, Burlington, MA 01803.

[9] IEEE COMPUTER. *Executable Object Modeling with Statecharts , Vol. 30, No. 7. .*, 1997.

[10] Bruce Powel. Douglass. *Doing Hard Time: Using Object Oriented Programming and Software Patterns in Real Time Applications.* Addison-Wesley, Reading, MA., 1998.

[11] Bruce Powell Douglass. *Real-Time UML,Efficient Objects for Embedded Systems.* Addison-Wesley-Longman, Spring, 1997.

[12] Matt Bishop George Fink. Property-based testing;a new approach to testing for assurance. In *Technical Report CSE-95-15*, 1996.

[13] Rance Cleaveland Gerald Litten, Michael von der Beeck. *A Compositional Approach to Statechart Semantics.* Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA.

[14] D. Harel. Statecharts : A visual formalism for complex systems. In *Science of Computer Programming vol.8*, 1987.

[15] Pankaj Jalote. *An Integrated Approach To Software Engineering.* Narosa Publishing House, New Delhi, 1999.

[16] James J.Odell. *Advanced Object-Oriented Analysis and Design Using UML.* Cambridge University Press and SIGS Books, New York, 1998.

[17] F. Levi. A process language for statecharts. In *Proceedings of LOMAPS'96, Springer LNCS vol 1192, pp 388–403*, 199.

[18] K. Lieberherr and I. Holland. *Assuring Good style for Object Oriented Programs.* IEEE Software, ., 1989.

[19] Information Processing Limited(IPL). Testing state machines with adatest and cantata. In *Softwre Testing,White papers*, 1999.

[20] Brian Marick. *The Craft Of Software Engineering,Subsystem Testing including Object-Based Testing and Object-Oriented Testing.* Prentice Hall PTR, New Jersey, 1995.

[21] Bertrand. Meyer. *Object-Oriented Software Construction, Second Edition.* Prentice Hall, Upper Saddle River, NJ., 1997.

[22] Pierre-Alain Muller. *Instant UML.* Wrox Press Ltd., Canada, 1999.

[23] A. Peron. Statecharts, transition structures and transformations. In *Proceedings of TAPSOFT'95, Springer LNCS vol 915, pp 454–468*, 1995.

[24] A. Peron and A. Maggilio-Schettini. Transitions as interrupts : A new semantics for timed statecharts. In *Proceedings of TACS'94, Springer LNCS vol 789, pp 806-821*, 1994.

[25] Shari Lawrence Pfleeger. *Software Engineering Theory and Practice.* Prentice Hall, New Jersey, 1998.

[26] Tsun S.Chow. Testing software design modeled by finite-state machines. In *IEEE Transactions on Software Engineering*, 1978.

[27] Bikram Sengupta. *On the Sematics of Statecharts.* Department of Computer Science, State University of New York al.Stony Brook, NY 11794 .

[28] Roger S.Pressman. *Software Engineering,Fourth Edition.* McGraw-Hill Companies,Inc., New York, 1997.